Text-Suffix-Fragment-Features

Eine neue Textrepräsentation für das Klassifizieren und Clustern natürlichsprachlicher Texte

Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Wirtschaftswissenschaften
(Dr. rer. pol.)

durch die Fakultät für Wirtschaftswissenschaften der
Universität Duisburg-Essen

vorgelegt von
MSc. Dipl.-Inform.(FH) Julia Salmen geb. Große-Brömer
aus Gladbeck

Gutachter: [Erstgutachter] Univ.-Prof. Dr. Zelewski
[Zweitgutachter] Univ.-Prof. Dr. Adelsberger

Tag der mündlichen Prüfung: 10.04.2013

Inhaltsverzeichnis

Abkürzungs- und Akronymverzeichnis

AAAI	Association for the Advancement of Artificial Intelligence
ACL	Association for Computational Linguistics
ACM	Association for Computing Machinery
ASCII	American Standard Code for Information Interchange
Aufl.	Auflage
Bd.	Band
BI	Business Intelligence
bspw.	beispielsweise
bzw.	beziehungsweise
C4.5	Name eines Algorithmus zum Aufbau eines Decision Trees
CA	Bundeststaat California
CART	Classification and Regression Trees
CCAT	Klassenbezeichnung aus dem RCV1-v2
CEC	Congress on Evolutionary Computation
CEUR	Center for European Union Research
CIKM	Conference on Information and Knowledge Management
CPM	Combinatorial Pattern Matching
CRC	Chemical Rubber Company
DC	District of Columbia
d.h.	das heißt
DNA	deoxyribonucleic acid (deutsch: Desoxyribonukleinsäure)

DOM	Document Object Model
DT	Decision Tree
ECAT	Klassenbezeichnung aus dem RCV1-v2
ECML	European Conference on Machine Learning
etc.	et cetera
EURASIP	European Association for Signal Processing
e.V.	eingetragener Verein
f.	folgende
FL	Bundesstaat Florida
GAAC	group-average agglomerative clustering
GCAT	Klassenbezeichnung aus dem RCV1-v2
GSA	generalisiertes Suffix Array
HLT	Human Language Technologies
Hrsg.	Herausgeber
IBM	International Business Machines Corporation
ICALP	International Colloquium on Automata, Languages and Programming
ICDM	International Conference on Data Mining
ICECT	International Conference on Electronics Computer Technology
ICML	International Conference on Machine Learning
ID	Identifizierer
ID3	Name eines Algorithmus zum Aufbau eines Decision Trees
IDC	International Data Corporation
IEEE	Institute of Electrical and Electronics Engineers
Inc.	Incorporated

IUI	Intelligent User Interfaces
Jg.	Jahrgang
KDD	Knowledge Discovery and Data Mining
KISS	Korea Information Science Society
kNN	k-Nearest-Neighbour
LLC	Limited Liability Company
LNAI	Lecture Notes in Artificial Intelligence
MA	Bundesstaat Massachusetts
MCAT	Klassenbezeichnung aus dem RCV1-v2
MIT	Massachusetts Institute of Technology
NB	Naive Bayes
NCD	Normalized Compression Distance
NGD	Normalized Google Distance
NIST	National Institute of Standards and Technology
NJ	Bundesstaat New Jersey
NLTK	Natural Language Toolkit
NP	nichtdeterministisch-polynomiell
NY	Bundesstaat New York
PA	Bundesstaat Pennsylvania
PAKDD	Pacific-Asia Conference on Knowledge Discovery and Data Mining
PAKM	Practical Aspects of Knowledge Management
PDF	Portable Document Format
PODS	Symposium on Principles of Database Systems
RCV1-v1	Reuters Corpus Volume 1 Version 1

RCV1-v2	Reuters Corpus Volume 1 Version 2
RI	Rand Index
RSS	residual sum of squares
S.	Seite
SAC	Symposium on Applied Computing
SGML	Standard Generalized Markup Language
SIAM	Society for Industrial and Applied Mathematics
SIGACT	Special Interest Group on Algorithms and Computation Theory
SIGIR	Special Interest Group on Information Retrieval
SIGKDD	Special Interest Group on Knowledge Discovery and Data Mining
SIGMOD	Special Interest Group on Management of Data
SODA	Symposium on Discrete Algorithms
SPIRE	String Processing and Information Retrieval
STOC	Symposium on the Theory of Computing
SVM	Support Vector Machine
TDWI	The Data Warehouse Institute
tf-idf	term frequency-inverse document frequency
TREC	Text Retrieval Conference
TSF	Text-Suffix-Fragment
u.a.	und andere
UML	Unified Modelling Language
Vgl.	Vergleiche
VLDB	Very Large Data Base Endowment Inc.
W3C	World Wide Web Consortium
WAE	Workshop on Algorithm Engineering
XML	Extensible Markup Language

Symbolverzeichnis

Berechnungsvorschriften

$\binom{n}{k}$	Binomialkoeffizient 'k aus n'[1]
$B(L_j \mid d_i)$	Bewertung einer Klasse L_j für das Dokument d_i
$\cos(x)$	Cosinusfunktion
$dm(x, y)$	Berechnung der Distanz zwischen x und y mit dem Distanzmaß dm
Entropie(S)	Entropie
Entropie(S_v)	Entropie der Menge der Trainingsbeispiele mit dem Wert v für das Feature f_r
$\exp(x)$	Exponentialfunktion
$\vec{f}_{d_i} \cdot \vec{f}_{d_{i'}}$	Skalarprodukt der Feature-Vektoren der Dokumente d_i und $d_{i'}$
Gain(S, f_r)	Information Gain des Features f_r bezogen auf die Teilmenge S der Trainingsdaten
$h(L_i, \omega_j)$	Anzahl der Dokumente in der zugewiesenen Klasse oder dem zugewiesenen Cluster ω_j, die zur vordefinierten Klasse L_i gehören
$L_{j_{d_i}} = \arg\max_{L_j \in L} funktion$	Klasse L_j ist die Klasse aus L, die aufgrund der $funktion$ für d_i eine maximale Bewertung erhält; das Dokument d_i wird

1 Für weitere Erläuterungen siehe Fußnote 2 auf S. 74.

	vom Naive-Bayes-Klassifizierer der Klasse L_j zugeordnet
$\ln(x)$	Logarithmusfunktion; natürlicher Logarithmus
$\log_2(x)$	Logarithmusfunktion zur Basis 2
$n!$	Fakultät von n
$P(A)$	Wahrscheinlichkeit für ein Ereignis A
$P(A\|B)$	Wahrscheinlichkeit für das Eintreten des Ereignisses A, wenn Ereignis B eingetreten ist
$P(f_r = 0\|L_j)$	Wahrscheinlichkeit, dass ein Feature f_r in der Klasse L_j, also in mindestens einem Trainingsdokument d_a, welches zur Klasse L_j gehört, nicht enthalten ist
$P(f_r = 1\|L_j)$	Wahrscheinlichkeit, dass ein Feature f_r in der Klasse L_j, also in einem Trainingsdokument d_a, welches zur Klasse L_j gehört, enthalten ist
$\text{score}(L_j, d_i)$	Punktzahl für die Klasse L_j, um ihr Dokument d_i zuzuweisen, wenn sie aufgrund der k nächsten Nachbarn die höchste Punktzahl erhält

Griechische Buchstaben

$\alpha \prec \beta$	Suffix α ist lexikografisch kleiner als Suffix β
$\alpha \succ \beta$	Suffix α ist lexikografisch größer als Suffix β
β	Gewichtungsfaktor im F_β-Maß

Γ	Symbol für den Rand Index für die Formel aus dem Originalpaper
γ	Zielfunktion beim Clustern
λ	Glättungsparameter im multinomialen Modell für einen Naive-Bayes-Klassifizierer
$\vec{\mu}$	Vektor des Clustermittelpunkts
Φ	Klassifizierer, der die Zielfunktion approximiert
$\Phi : D \times L \rightarrow \{true, false\}$	Funktionsbeschreibung der Funktion Φ mit dem Definitionsbereich $D \times L$ und dem Wertebereich $\{true, false\}$
$\breve{\Phi}$	unbekannte Zielfunktion in der Klassifizierung
χ^2	Chi-Quadrat-Verteilung
Ω	Menge der Cluster, ein Clustering
$\widehat{\Omega}$	Menge der möglichen Clusterings
ω_i	Bezeichnung für ein Cluster einer Menge von Clustern Ω
$\|\omega\|$	Anzahl der im Cluster ω vorhandenen Dokumente

Lateinische Buchstaben

$a[i]$	Element an Position i des Arrays a
$a[i][j]$	Element in Zeile i und Spalte j des Arrays a
b	Achsenabschnitt der Geraden, die durch eine SVM erzeugt wird

BL-%	Baseline mit randomisierter Zuordnung
BL-C152	Baseline für die Klasse mit den meisten Trainingsdokumenten im Experiment Klass1TV
BL-I81402	Baseline für die Klasse mit den meisten Trainingsdokumenten im Experiment Klass1IV
BL-USA	Baseline für die Klasse mit den meisten Trainingsdokumenten im Experiment Klass1RV
C	Parameter in einer Implementierung einer SVM
c	Anzahl der Cluster einer Menge von Clustern Ω, das entspricht $\|\Omega\|$
D	Dokumentsammlung, eine Menge von Dokumenten
d_i	Dokument d_i
$\|D\|$	Anzahl der Dokumente der Menge D
F	Featuremenge F einer Dokumentsammlung
F_β	F_β-Maß zur Evaluation einer erzeugten Klassifikation oder eines erzeugten Clusterings
F_{d_i}	Menge der Features des Dokuments d_i
F'_{d_i}	Menge der potenziellen Features des Dokuments d_i
FN	Anzahl der Dokumente, die bei der Evaluation einer erstellten Klassifikation oder eines erzeugten Clusterings als *false negative* evaluiert werden
FP	Anzahl der Dokumente, die bei der Evaluation einer erstellten Klassifikation oder eines erzeugten Clusterings als *false positive* evaluiert werden

$\vec{f}_{d_i}$	Feature-Vektor des Dokuments d_i
$f_{r_{d_a}}$	Wert des Features f_r, den es in Dokuments d_a hat
$\|F\|$	Gesamtanzahl der Features in der Featuremenge F
$\|\vec{f}_{d_i}\|$	euklidische Norm also die Länge des Feature-Vektors $\vec{f}_{d_i}$
k	Anzahl der Nachbarn im k-Nearest-Neighbour-Algorithmus; aber auch Anzahl der zu erzeugenden Cluster im k-Means-Algorithmus, diese wird in der vorliegenden Arbeit jedoch als c bezeichnet
L	Menge der Klassen einer vorgegebenen Klassifikation
L/S	Bezeichnung des Typs nach Ko und Aluru für das Endezeichen einer Zeichenkette
$\|L\|$	Anzahl der Klassen der Menge L
$\|L_j\|$	Anzahl der Dokumente der Klasse L_j
m	Anzahl der Listen, die bei der Erzeugung eines Suffix Arrays nach Ko und Aluru benötigt werden
$Prec$	Genauigkeit, vom Englischen *precision*
Rec	Trefferquote, vom Englischen *recall*
RSS	Summe der quadratischen Distanzen der Feature-Vektoren zu ihrem Clustermittelpunkt in einem Clustering
RSS_{ω_j}	Summe der quadratischen Distanzen der Feature-Vektoren zu ihrem Clustermittelpunkt in Cluster ω_j

S	betrachtete Teilmenge der Trainingsdaten beim Aufbau eines Entscheidungsbaums
ST_1	Suffix Tree, der Suffix 1 des Textes enthält
$\|S_{L_j}\|$	Anzahl der Trainingsdaten der Teilmenge S mit Klassenzuordnung L_j
T	ein Text T
T'	Zeichenkette, die zu Teilzeichenketten aus T korrespondiert und sortiert wird, um T sortieren zu können
TE	Menge der Testdaten für einen Klassifizierer
TN	Anzahl der Dokumente, die bei der Evaluation einer erstellten Klassifikation oder eines erzeugten Clusterings als *true negative* evaluiert werden
TP	Anzahl der Dokumente, die bei der Evaluation einer erstellten Klassifikation oder eines erzeugten Clusterings als *true positive* evaluiert werden
TR	Menge der Trainingsdaten für einen Klassifizierer
$T[1..i]$	Präfix des Textes T, das an Position i endet
$T[i]$	Suffix des Textes T, das an Position i im Text beginnt und am Ende des Textes endet; verkürzte Schreibweise für $T[i..\|T\|]$
$T[i..j]$	Teiltext von T beginnend an Position i im Text und endend an Position j im Text
$T[i..\|T\|]$	Suffix des Textes T, das an Position i im Text beginnt und am Ende des Textes endet

$T[i][j]$	Darstellung des Zugriffs auf das Zeichen an Position j im Suffix $T[i]$
T_{konk}	eine Konkatenation mehrerer Texte zu einem neuen Text
$\|T\|$	Länge eines Textes T
$\|TR\|$	Anzahl der Trainingsdokumente
$\|V_{f_r}\|$	Anzahl der Werte des Features f_r
$\vec{w}$	Vektor, der senkrecht zur Geraden ist, die durch eine SVM erzeugt wird
$\vec{x}$	Stützvektoren der Geraden, die durch eine SVM erzeugt wird
$\overline{x}_{\text{harmonisch}}$	harmonischer Mittelwert
$y[i]$	Darstellung des Zugriffs auf Element i der Liste y
$\mathbb{Z}$	Menge der ganzen Zahlen

Produkt- und Summenzeichen

$\prod_{i=1}^{n} x_i$	Multiplikation über Variable x von x_1 bis x_n
$\sum_{i=1}^{n} x_i$	Summe über Variable x von x_1 bis x_n
$\sum_{x \in M} x$	Summe über alle x aus der Menge M
$\sum_{i=1}^{n} x_i$	Summe über Variable x von x_1 bis x_n

Sonstige

$!=$	Darstellung für $\neq$ in den Abbildungen in der vorliegenden Arbeit
$\neq$	ungleich im Text und in den Algorithmen der vorliegenden Arbeit
$<$	kleiner
$<=$	Darstellung für $\leq$ in den Abbildungen in der vorliegenden Arbeit
$\leq$	kleiner gleich im Text und in den Algorithmen der vorliegenden Arbeit
$=$	Zuweisung eines Wertes
$==$	Vergleichsoperator, der angibt, ob beide Seiten des doppelten Gleichheitszeichens gleich sind, jedoch keine Wertzuweisung vornimmt
$>$	größer
$>=$	Darstellung für $\geq$ in den Abbildungen in der vorliegenden Arbeit
$\geq$	größer gleich im Text und in den Algorithmen der vorliegenden Arbeit
\$	US-Dollar; bei Suffix Trees und Suffix Arrays: Endezeichen eines Textes
$\cap$	Durchschnitt zweier Mengen
$\cup$	Vereinigung zweier Mengen
$\emptyset$	leere Menge
$\in$	ist Element von
$\subset$	echte Teilmenge
$A \times B$	kartesisches Produkt der Mengen A und B

$\langle d_i, L_j \rangle \in D \times L$	alle Paare aus Dokument und Klasse, die dem kartesischen Produkt der Mengen D und L entstammen
∞	Unendlichkeit
$\Rightarrow$	daraus folgt
$\rightarrow$	innerhalb einer Funktionsbeschreibung wird dadurch die Zuordnung eines Elements des Definitionsbereichs zu einem Element des Wertebereichs dargestellt
$\wedge$	und
$\lvert \left\{ \vec{f}_{d_a} \lvert a = 1..\lvert D \rvert \wedge d_a \in L_j \wedge f_{r_{d_a}} = 0 \right\} \rvert$	Anzahl der Vektoren aller Dokumente d_a aus D für die gilt: $d_a \in L_j$ und der Wert des betrachteten Features f_r ist im betrachteten Dokument d_a gleich 0
$\lvert \left\{ \vec{f}_{d_a} \lvert a = 1..\lvert D \rvert \wedge d_a \in L_j \wedge f_{r_{d_a}} = 1 \right\} \rvert$	Anzahl der Vektoren aller Dokumente d_a aus D für die gilt: $d_a \in L_j$ und der Wert des betrachteten Features f_r ist im betrachteten Dokument d_a gleich 1
$A.length$	letzte Position eines Arrays A
$\frac{\lvert S_v \rvert}{\lvert S \rvert}$	Anzahl der Trainingsbeispiele aus S mit Wert v für das Feature f_r, dividiert durch die Anzahl der Trainingsbeispiele in S

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Realproblem

Wesentliche Teile der Daten, die in Unternehmen für die tägliche Arbeit benötigt werden, entstammen natürlichsprachlichen Texten[1], die in verschiedenen Formen, zum Beispiel als Dokumente oder E-Mails, vorliegen können. Diese Daten nennt man *semi-* oder *unstrukturierte* textuelle Daten [2].
Das Problem im Umgang mit diesen semi- oder unstrukturierten textuellen Daten ist, dass sie sich nicht ohne Weiteres mit den computergestützten Datenmanagementsystemen in einem Unternehmen verarbeiten lassen. Diese vorhandenen Systeme sind darauf ausgelegt, strukturierte Daten zu speichern und zu Abfragen gefundene Daten zurückzuliefern. Die semi- oder unstrukturierten textuellen Daten müssten demnach zunächst in eine strukturierte Form gebracht werden, um mit den vorhandenen Systemen nutzbar zu sein, oder es müssten neue Systeme oder Ergänzungen zu den alten Systemen dazukommen, die die semi- oder unstrukturierten textuellen Daten so weiterverarbeiten, dass sie ebenfalls in Unternehmensprozessen zur Problemlösung

1 Vgl. Tan (1999), S. 65. Der genaue Anteil dieser Daten an den Gesamtdaten in einem Unternehmen ist nicht genau bekannt. Zwar verweist Tan (1999), S. 65, auf einen Anteil von 80%, belegt diesen aber nicht. Grimes (2008), S. 1 f. (Seitennummerierung der Verfasserin, da Webdokument), kommt nach einer genaueren Analyse zu dem Schluss, dass dieser Anteil nie wirklich belegt wurde, und zitiert eine Studie, die einen Anteil von 53% an semi- und unstrukturierten Daten ermittelt, vgl. Russom (2007), S. 8. Für Grimes ist nicht der genaue Anteil entscheidend, sondern die Tatsache, dass diese Daten einen großen Stellenwert in den Unternehmen besitzen. Der Begriff „natürlichsprachliche Texte“ wird in der vorliegenden Arbeit synonym für „natürlichsprachliche Dokumente“, „Dokumente“ und „textuelle Daten“ verwendet.

2 Im Allgemeinen werden *strukturierte*, *semi-strukturierte* und *unstrukturierte* Daten voneinander unterschieden. Laut Grossman u.a. (2004), S. 211, ist eine formale Definition von strukturierten Daten schwierig. Sie zeichnen sich durch einfache Wiedererkennbarkeit der Struktur aus und tauchen wiederholt in der gleichen Form auf. Weiss u.a. (2005), S. 2, vergleichen strukturierte Daten mit einer Tabelle. Beispiele für strukturierte Daten wären Adressen.

Semi-strukturierte Daten dagegen bestehen aus einem strukturierten Teil und einem unstrukturierten Teil. Abiteboul u.a. (2000), S. 13, beschreiben semi-strukturierte Daten als Daten, die keinen vordefinierten Datentyp haben, sondern eine Beschreibung der Daten zusätzlich zu den eigentlichen Daten enthalten. Dabei kann es sich z.B. um ein XML-Dokument mit einem natürlichsprachlichen Text oder eine E-Mail handeln. Hier wären die XML-Tags bzw. die Header-Informationen der E-Mail der strukturierte Teil, während der eigentliche Inhalt des Dokuments der unstrukturierte Teil wäre.

Unstrukturierte Daten besitzen keine Zusatzinformationen, aus denen sich eine Struktur der Daten ablesen ließe. Diese Zusatzinformationen müssen explizit extrahiert werden, um verarbeitet werden zu können. Beispiele für unstrukturierte Daten wären Texte, die nicht in XML oder einem ähnlich strukturierten Format vorliegen, oder auch Bilder ohne zusätzliche Metadaten.

beitragen können. Die semi- oder unstrukturierten Daten liegen zumeist bereits in digitaler Form vor. Aus diesem Grund bietet sich eine Aufbereitung der textuellen Daten durch den Computer an, um dem Menschen einen automatischen Umgang mit den betreffenden Daten zu ermöglichen. Diese Aufbereitung dient dazu, eine in den textuellen Daten implizit vorhandene, aber nicht explizit ausgezeichnete Struktur zu ermitteln und sie explizit zu machen, damit sich die Daten in den vorhandenen Systemen - oder den neuen Komponenten - auf ähnliche Art und Weise weiterverarbeiten lassen wie strukturierte Daten. Das bedeutet, es ist nicht entscheidend, ob die Daten zunächst aufbereitet werden, um in vorhandenen Systemen genutzt zu werden, oder ob sie direkt von neuen Systemen oder Komponenten aufbereitet werden, sondern die Aufbereitung an sich ist der wesentliche Aspekt.

Wird keine Aufbereitung der textuellen Daten durchgeführt, kann das Nachteile für die betroffenen Unternehmen haben:

1. Wichtige Informationen[1] bleiben unberücksichtigt.

1 Die Begriffe „Daten“ und „Informationen“ werden im Sprachgebrauch häufig synonym verwendet. Trotzdem scheinen auch Unterschiede zwischen ihnen zu bestehen. Das Problem ist die Definition dieses Unterschieds, der oftmals nicht exakt verdeutlicht wird, vgl. Boisot u.a. (2004), S. 44. In der Literatur finden sich häufig die folgenden Definitionen der beiden Begriffe:

- *Daten* beschreiben Ereignisse der Welt, also physikalische messbare Veränderungen, vgl. Boisot u.a. (2004), S. 46. Es sind diskrete, objektive Fakten, die gespeichert werden können, jedoch für sich allein betrachtet nichts über ihre Wichtigkeit aussagen, vgl. Davenport u.a. (2000), S. 2 (Seitennummerierung durch die Verfasserin, da Internetdokument). Auch Logan (2012), S. 83, definiert Daten als Fakten, die keine Struktur aufweisen und lediglich als Bausteine für Information zu verstehen sind. Gopal u.a. (2011), S. 728, halten den Begriff „Daten“ für einen sehr weit gefassten Terminus, definieren diesen aber nicht weiter.
- *Information* ist laut Boisot u.a. (2004), S. 47, eine Verbindung zwischen Daten und einem datenverarbeitenden Agenten - einem Menschen oder einer Maschine. Davenport u.a. (2000), S. 3 (Seitennummerierung durch die Verfasserin, da Internetdokument), beschreiben Information als Nachricht von einem Sender an einen Empfänger. Information entsteht also durch Daten, die durch den Empfänger mit Bedeutung versehen werden. Als Beispiele nennen Davenport u.a. (2000), S. 3 (Seitennummerierung durch die Verfasserin, da Internetdokument), Daten einen Kontext hinzufügen, sie zu klassifizieren, sie in Berechnungen einfließen zu lassen, sie zu korrigieren oder zusammenzufassen. Logan (2012), S. 83, beschreibt Information als strukturierte Daten. Durch diese Struktur erhalten die Daten eine Bedeutung und einen Kontext.

  Aus dem Unterschied zwischen den Definitionen wird deutlich, dass Information aus der *Interpretation* von Daten gewonnen wird, vgl. Boisot u.a. (2004), S. 55, und dass unterschiedliche Agenten unterschiedliche Information aus den gleichen Daten gewinnen können, vgl. Boisot u.a. (2004), S. 54.

  Eine etwas andere Definition der beiden Begriffe findet sich bei Tuomi (1999), S. 4 f. Er führt aus, dass die Basis der oben gegebenen Definitionen ein Prozess ist, bei dem Schritt für Schritt Bedeutung hinzugefügt wird, also eine Konvertierung von einfach - Daten - bis komplex - Wissen - stattfindet, vgl. Tuomi (1999), S. 2. Der Autor selbst vertritt jedoch die Meinung, dass die Welt selbst, also auch alle Beobachtungen, nur interpretiert existiert und demzufolge keine Daten ohne Bedeutung vorhanden sind, vgl. Tuomi (1999), S. 2. Für ihn existieren Daten nur dann, wenn sie zuvor von jemandem - unter Berücksichtigung seines Wissens - artikuliert werden, das ist Information, und diese so abgebildet wird, dass sie gespeichert werden kann, das sind Daten. Für diesen Speichervorgang muss die Information in der Struktur der Daten abgelegt werden, vgl. Tuomi (1999), S. 5.

  In der vorliegenden Arbeit wird die Definition von Tuomi für die beiden Begriffe „Daten“ und „Information“ zu Grunde gelegt, ohne aber die anderen Definitionen zu vernachlässigen. Das wird von Tuomi ebenfalls so gehandhabt, da er ein Beispiel beschreibt, das genau die oben beschriebene Situation der Verarbeitung von textuellen Daten wiedergibt. In diesem Beispiel, vgl. Tuomi (1999), S. 8 f., beschreibt er eine Praxissituation, in der jemand sein Wissen artikuliert und gespeichert hat, bspw. in Form von Dokumenten auf einem Computer. Diese Dokumente bilden Daten, auf die jemand anders zugreift. Dieser andere Agent versucht nun, aus den Daten ihre ursprüngliche Bedeutung unter Berücksichtigung seines Wissens wiederherzustellen. Bezogen auf die beschriebene Situation, dass textuelle Daten nicht aufbereitet werden, ergibt sich somit, dass die Bedeutung, also in diesem Fall die Information, die in ihnen enthalten ist, unberücksichtigt bleibt.

2. Es entsteht ein Zeitverlust - und damit auch ein materieller Verlust - bei der Bearbeitung von unwichtigen Daten.

Zum ersten Punkt ist zu sagen, dass trotz einer so genannten „Informationsflut“[1], also dem übermäßigen Vorhandensein von Informationen und dem einfachen und schnellen Zugriff auf diese Informationen, insbesondere das schnelle Finden problemrelevanter Inhalte und Zusammenhänge noch unzureichend unterstützt[2] wird. Das führt nach Rao (2003)[3]

1. zu unnötigen Kosten, die durch die Zeit entstehen, die Mitarbeiter des Unternehmens damit zubringen, nach Informationen zu suchen und diese zu bewerten,

2. zu einem weiteren Anwachsen der vorhandenen Informationen, da viele textuelle Informationen, die nicht gefunden werden, nochmal erzeugt werden, anstatt sie wiederzuverwenden, und

3. zu eventuell falschen Entscheidungen oder entgangenen Möglichkeiten durch das Nicht-Berücksichtigen vorhandener Informationen.

Eine IDC-Studie[4] belegt diese Punkte mit konkreten Zahlen. In drei Szenarien werden die Kosten, die einem Unternehmen mit 1.000 so genannten „Knowledge Management Workers“[5] entstehen, geschätzt. Dabei werden folgende Kosten geschätzt:

1. Die verschwendete Zeit, die diese 1.000 Mitarbeiter mit dem Suchen nach Informationen verbringen, kostet das Unternehmen 2,5 Millionen US-Dollar pro Jahr.
   Hier wird Folgendes zu Grunde gelegt[6]:

   - Jeder Mitarbeiter erhält ein Jahresgehalt von 80.000 $.
   - Jeder Mitarbeiter verbringt pro Woche 1,25 Stunden mit der vergeblichen Suche nach Informationen.

1 Krcmar (2005), S. 52-54. In der Literatur wird Informationsflut auch als „Information Overload“ bezeichnet, vgl. Stergios (2010), S. 1 (Seitennummerierung der Verfasserin, da Webdokument).
2 Vgl. Furguson (2005), S. 44; Bruce u.a. (2004), S. 11 (Seitennummerierung der Verfasserin, da Webdokument); Rao (2003), S. 29.
3 Vgl. Rao (2003), S. 29.
4 Vgl. Feldman u.a. (2000), S. 7-9.
5 Es existiert keine einheitliche Definition, was man unter einem „Knowledge Management Worker“ genau versteht. Davenport u.a. (1998), S. 108-112, erläutern, dass der Umgang mit Wissen eine Aufgabe ist, die jeder Mitarbeiter eines Unternehmens erfüllen sollte, auch wenn sein eigentlicher Aufgabenbereich ein anderer ist. Sie stellen weiterhin spezifische Rollen für Mitarbeiter vor, die hautpsächlich mit der „Verwaltung“ von Wissen betraut sind.
6 Vgl. Feldman u.a. (2000), S. 7.

- Ein Arbeitsjahr hat 52 Wochen mit 40 Stunden wöchentlicher Arbeitszeit pro Mitarbeiter.

Daraus resultiert:

$$\begin{aligned}\text{Kosten des Unternehmens} \\ \text{pro Mitarbeiter pro Stunde} &= \frac{80.000\ \$}{\text{Jahr und Mitarbeiter}} * \frac{\text{Jahr}}{52\ \text{Wochen}} * \\ &\quad \frac{\text{Woche}}{40\ \text{Stunden}} \\ &= \frac{38,46153846\ \$}{\text{Stunde und Mitarbeiter}}\end{aligned}$$

$$\begin{aligned}\text{Kosten des Unternehmens} \\ \text{pro Jahr für die vergebliche Suche} &= \frac{38,46153846\ \$}{\text{Stunde und Mitarbeiter}} * \frac{1,25\ \text{Stunden}}{\text{Woche}} * \\ &\quad \frac{52\ \text{Wochen}}{\text{Jahr}} * 1.000\ \text{Mitarbeiter} \\ &= \frac{2.500.000\ \$}{\text{Jahr}}\end{aligned}$$

2. Die Kosten, die entstehen, weil bereits vorhandene Informationen im Unternehmen nochmals erzeugt werden, belaufen sich bei 1.000 Mitarbeitern auf 5 Millionen US-Dollar pro Jahr.[1]

3. Die Opportunitätskosten des Unternehmens lassen sich schwieriger bestimmen, werden hier aber mit mehr als 15 Millionen US-Dollar pro Jahr angegeben. Bei diesem Szenario wird Folgendes zu Grunde gelegt[2]:

   - Der „*Revenue*“[3] des Unternehmens pro Mitarbeiter pro Stunde beträgt 240 $.
   - Jeder Mitarbeiter verbringt 1,25 Stunden pro Woche mit der vergeblichen Suche nach Informationen.
   - Ein Arbeitsjahr hat 52 Wochen.

1 Hier wird zu Grunde gelegt, dass durchschnittlich 5.000 US-Dollar pro Jahr pro Kopf an Kosten für das Duplizieren von Informationen entstehen, vgl. Feldman u.a. (2000), S. 7 f.
2 Vgl. Feldman u.a. (2000), S. 8.
3 Feldman u.a. (2000), S. 8. Aus der Quelle lässt sich nicht bestimmen, was sich genau hinter dem verwendeten Begriff *revenue* verbirgt. Im Deutschen wird *revenue* häufig mit „Erlös“ übersetzt, vgl. Klein u.a. (2008), S. 5.

Daraus resultiert:

$$\text{Opportunitätskosten} = \frac{240\ \$}{\text{Stunde und Mitarbeiter}} * \frac{1,25\ \text{Stunden}}{\text{Woche}} * \frac{52\ \text{Wochen}}{\text{Jahr}} * \frac{1.000\ \text{Mitarbeiter}}{} = \frac{15.600.000\ \$}{\text{Jahr}}$$

Während sich der erste Punkt, dass wichtige Informationen unberücksichtigt bleiben, eher auf das Nicht-Finden von relevanten Informationen bezieht, betrifft der zweite Punkt, dass unwichtige Informationen bearbeitet werden, den Zeitpunkt *nach* dem Finden oder dem Erhalt von Informationen. Der entscheidende Punkt hier ist, dass, obwohl die Daten digital vorliegen, der Mensch trotzdem, wenn es sich wie hier um semi- oder unstrukturierte, textuelle Daten handelt, gezwungen ist, diese zu bearbeiten. Er muss sie lesen, um entscheiden zu können, ob die daraus gewonnenen Informationen brauchbar sind oder nicht. Durch die oben angesprochene Masse an semi- oder unstrukturierten Daten werden so auch Informationen bearbeitet, die unwichtig sind. Trotzdem wird in diese Informationsbearbeitung Zeit investiert.[1]
Ein Beispiel dafür liefern Zeldes u.a. (2007). Die Autoren beziehen sich auf mehrere bei Intel durchgeführte Erhebungen[2] über das E-Mail-Overload-Problem. In diesen Erhebungen ergab sich ein Anteil von 30% an überflüssigen E-Mails[3], die die Mitarbeiter erhielten, die aber trotzdem bearbeitet wurden, da sie nicht direkt als überflüssig erkannt wurden. In einem Rechenbeispiel ermitteln Zeldes u.a. (2007)[4] Kosten von 245 Millionen US-Dollar im Jahr bei einem Unternehmen mit 50.000 Mitarbeitern für die Bearbeitung von unwichtigen E-Mails. Zu Grunde gelegt wurde Folgendes[5]:

- Jeder Mitarbeiter benötigt 2 Stunden pro Woche, um seine unwichtigen E-Mails zu bearbeiten.[6]
- Die Arbeitskosten pro Stunde pro Mitarbeiter betragen 50 $.
- Ein Arbeitsjahr hat 49 Wochen.

1 In der vorliegenden Arbeit wird davon ausgegangen, dass unstrukturierte textuelle Daten vorliegen. Auch E-Mails werden als unstrukturierte Daten angesehen, da nur der unstrukturierte Teil, der Text, betrachtet wird, aber keine strukturierten Teile der E-Mails.
2 Vgl. Zeldes u.a. (2007), S. 6 (Seitennummerierung durch die Verfasserin, da Webdokument).
3 Vgl. Zeldes u.a. (2007), S. 3 (Seitennummerierung durch die Verfasserin, da Webdokument).
4 Vgl. Zeldes u.a. (2007), S. 8 (Seitennummerierung durch die Verfasserin, da Webdokument).
5 Vgl. Zeldes u.a. (2007), S. 8 (Seitennummerierung durch die Verfasserin, da Webdokument).
6 Vgl. Zeldes u.a. (2007), S. 6 (Seitennummerierung durch die Verfasserin, da Webdokument).

Daraus resultiert:

$$\begin{aligned}\text{Kosten des Unternehmens} \\ \text{für die Bearbeitung unwichtiger} \\ \text{E-Mails pro Jahr} &= \frac{\text{2 Stunden}}{\text{Woche und Mitarbeiter}} * \frac{\text{50 \$}}{\text{Stunde}} * \frac{\text{49 Wochen}}{\text{Jahr}} * \\ &\quad \text{50.000 Mitarbeiter} \\ &= \frac{\text{245.000.000 \$}}{\text{Jahr}}\end{aligned}$$

Diese Beispiele zeigen sehr deutlich, dass sich das Nicht-Aufbereiten textueller Daten in jedem Fall, egal ob es um die Suche nach Informationen oder den Zeitaufwand für die Bearbeitung eigentlich unwichtiger Informationen geht, nachteilig auf Unternehmen auswirken kann.

Das Erstellen, die Wartung und die Verwendung von Klassen[1] stellt ein wesentliches Instrument im Umgang mit textuellen Daten dar.[2] Die Erzeugung einer Klassifikation[3] aus den vorliegenden textuellen Daten durch eine Software wird als *Clustern* bezeichnet und das Ergebnis eines Cluster-Vorgangs als *Clustering*.[4] Neben automatisiert erzeugten *Klassifikationen* existieren auch manuell erzeugte Klassifikationen. Den Klassen der Klassifikation können neue Dokumente durch eine *Klassifizierung*[5] hinzugefügt werden.[6]

Eine computergestützte Erzeugung einer Klassifikation durch Clustern vorhandener textueller Daten und die anschließende Klassifizierung neu hinzukommender textueller Daten könnte in den zuvor beschriebenen Beispielszenarien eine Kostenersparnis für die Unternehmen bedeuten.[7] Wären bspw. die E-Mails der Mitarbeiter im zweiten Beispiel in einer Klassifikation aufgrund ihrer Wichtigkeit für den jeweiligen

1 Alternativ hier: Kategorien, Cluster. Eine präzise begriffliche Abgrenzung ist in einem späteren Kapitel enthalten. Im weiteren Verlauf des Kapitels wird Klasse stellvertretend für die beiden anderen Begriffe verwendet.

2 Vgl. Manning u.a. (2008), S. 235; Russell u.a. (2003), S. 845; Koller u.a. (1997), S. 170.

3 In Anlehnung an die DIN-Norm 32705 wird „Klassifikation“ als Bezeichnung für das Resultat der Erzeugung von Klassen verwendet, vgl. Deutsches Institut für Normung e.V.(1987), S. 5. Der Prozess der Zuordnung von Dokumenten zu dieser erzeugten Klassifikation als „Klassifizierung“, vgl. Deutsches Institut für Normung e.V.(1987), S. 8.

4 Siehe auch Kapitel 2.4 dieser Arbeit.

5 Die Begriffe *Klassifizierung* und *Klassifizieren* werden in der vorliegenden Arbeit synonym verwendet.

6 Siehe auch Kapitel 2.3 dieser Arbeit.

7 Natürlich entstehen durch den Einsatz einer Software ebenfalls Kosten. Diese werden in den folgenden Überlegungen nicht berücksichtigt, da sie nicht bekannt sind. Die Verfasserin geht jedoch davon aus, dass mehr Kosten durch den geringeren menschlichen Arbeitsaufwand bei der Bearbeitung unwichtiger E-Mails eingespart werden als Kosten durch die Software verursacht werden.

Mitarbeiter enthalten, so könnten neu hinzukommende E-Mails automatisch in die Klassen dieser Klassifikation einsortiert werden. Ließe sich dieser Vorgang automatisch durchführen und ergäbe sich, dass 100% der E-Mails korrekt klassifiziert wären, dann ginge den Mitarbeitern keine Zeit für die Bearbeitung für sie unwichtiger E-Mails verloren.
Obwohl in der Forschung bereits viele Algorithmen zur automatisierten Aufbereitung von textuellen Daten[1] entwickelt wurden, werden sie noch nicht verbreitet in den Unternehmen eingesetzt. Dafür gibt es zwei Gründe, die sich gegenseitig bedingen und die das Realproblem der vorliegenden Arbeit darstellen:

1. Es erweist sich als schwierig, Klassifikationen zu erstellen, sie anzuwenden und sie zu warten, da immer *Zusatzwissen*[2] nötig ist.
   Blumberg u.a. (2003) beschreiben es als die Notwendigkeit Mitarbeiter zu haben, die

   > „...both understand the organization's business and happen to have a degree in library science - a rare combination."[3]

   Robb (2004)[4] verdeutlicht ebenfalls, dass das Benutzen von Werkzeugen in diesem Bereich zumeist Spezialisten vorbehalten bleibt. Spangler u.a. (2007)[5] erläutern, dass die meisten Algorithmen in sehr artifiziellen Umgebungen getestet wurden und diese Algorithmen sich nicht so einfach an jeden Kontext anpassen lassen, da sie zu sehr auf den Bereich zugeschnitten sind, in dem sie erprobt wurden. Daraus folgt der zweite Punkt.

2. Die Qualität der erzeugten Klassifikationen ist bisher nicht überzeugend. So bemängeln Blumberg u.a. (2003)[6], dass die meisten Systeme nur einen Algorithmus implementieren, der nicht in jedem Fall mit zufriedenstellender Qualität klassifiziert. Auch Bayer (2010)[7] betont die Nicht-Ausgereiftheit von Textanalysetechniken und den Mangel an Systemen für Unternehmen, die vollständig mit unstrukturierten Daten umgehen können. Es würde eher experimentell

1 Darunter versteht man das Erzeugen eines Clusterings und das Klassifizieren von textuellen Daten.
2 Als *Zusatzwissen* versteht die Verfasserin in dieser Arbeit sowohl Wissen über die zu klassifizierenden oder zu clusternden Daten an sich, also über deren Inhalt und Aufbau, als auch technisches Wissen in Bezug auf die durchzuführenden Vorverarbeitungsschritte zur Datenaufbereitung sowie die Auswahl und Parametrisierung der Algorithmen.
3 Blumberg u.a. (2003), S. 44.
4 Vgl. Robb (2004), S. 2 (Seitennummerierung der Verfasserin, da Webdokument).
5 Vgl. Spangler u.a. (2007), S. 4.
6 Vgl. Blumberg u.a. (2003), S. 44.
7 Vgl. Bayer (2010), S. 13.

mit kleineren Einzelwerkzeugen innerhalb von Business-Intelligence-Systemen versucht, einen Mehrwert in Bezug auf Textanalysen zu schaffen.

1.2 Betriebswirtschafliche Desiderate

Betriebswirtschaftlich wünschenswert ist eine computergestützte Aufbereitung der natürlichsprachlichen Texte[1] mit erhöhter Qualität, die in einem ersten Schritt kein explizites Zusatzwissen auf Seiten der Mitarbeiter eines Unternehmens benötigt.

1.3 State of the art

Insbesondere, wenn es um den Umgang mit unstrukturierten textuellen Daten geht, wird eine automatische[2] Klassifizierung vorgeschlagen, um zusätzliche Daten über die Inhalte der textuellen Daten zu erzeugen[3], die eine Suche nach diesen textuellen Daten oder die Entscheidung, ob die Daten relevant sind oder nicht, erleichtern. Zu einer solchen automatischen Klassifizierung gehört eine Vorverarbeitung der Texte und schließlich die Zuordnung dieser Texte zu Klassen. Das setzt eine bereits vorhandene oder aber eine zu erzeugende Klassifikation voraus. Dies entspricht der typischen Vorgehensweise in der derzeitigen Literatur.[4]
Die These in dieser Arbeit ist, dass ein zentraler Schwachpunkt aktuell verwendeter Verfahren die starke Wortorientierung ist, die vorhandene syntaktische Strukturen und semantische Zusammenhänge auflöst.[5] Eine Klassifizierung und ein Clustern beruhen auf einem entsprechenden Algorithmus. Dieser erhält als Eingabe in den meisten Fällen Vektoren, die die Dokumente darstellen. Jede Position in einem sol-

1 Natürlichsprachliche Texte sind in der vorliegenden Arbeit alle Dokumente, die Text enthalten, der nicht durch Metadaten ausgezeichnet ist. Insbesondere werden für die computergestützte Aufbereitung elektronisch vorliegende Dokumente benötigt. In der vorliegenden Arbeit werden die Begriffe „natürlichsprachlicher Text“, „Dokument“, „Text“ und „textuelle Daten“ synonym verwendet. Diese Texte sind in der vorliegenden Arbeit unstrukturierte Daten. Auch E-Mails, die eigentlich semi-strukturiert sind, werden als unstrukturiert behandelt, da nur der enthaltene Text berücksichtigt wird.

2 In dieser Arbeit wird unter einer automatischen Klassifizierung die Durchführung der Klassifizierung durch eine Software verstanden. Diese Klassifizierung erfolgt in einem ersten Schritt autonom durch die Software. Es soll dem Menschen aber möglich sein, Nachbesserungen vorzunehmen. Ist das der Fall, spricht man von einer teil-automatischen Klassifizierung. Alle Betrachtungen in dieser Arbeit beziehen sich jedoch, außer es ist explizit etwas anderes angegeben, auf eine autonom durch eine Software durchgeführte Klassifizierung.

3 Vgl. Rao (2003), S. 29.

4 Vgl. Kehagias u.a. (2003), S. 228; Sebastiani (1999), S. 5 (Seitennummerierung der Verfasserin, da Webdokument); Dörre u.a. (1999), S. 399 f.; Poulos u.a. (2005), S. 1 (Seitennummerierung der Verfasserin, da Webdokument); Yang (2006), S. 93.

5 Vgl. Bengio (2003), S. 1138.

chen Vektor stellt ein so genanntes Feature dar.[1] Diese Features sind in der aktuellen Literatur in den meisten Fällen Einzelworte.[2] Das bedeutet, der textuelle Inhalt eines Dokuments wird in seine einzelnen Worte[3] zerlegt, dies ist die Menge der potenziellen Features des Dokuments. Aus dieser Menge von potenziellen Features werden die zu verwendenden Features ermittelt und deren Vorkommen, Vorkommenshäufigkeit oder gewichtete Vorkommenshäufigkeit bilden den Feature-Vektor des Dokuments.[4]

Um die Menge der zu verwendenden Features aus der Menge der potenziellen Features zu ermitteln, können Worte aussortiert werden - so genannte Stoppworte[5] - und ein Stemming[6] der Worte vorgenommen werden. Die Worte, die danach noch zur Menge der potenziellen Features gehören, sind die zu verwendenden Features des Dokuments.

Durch diese Vorgehensweise wird der Inhalt des Dokuments zerstückelt und auf - gemessen an der Gesamtanzahl aller in einem einzelnen Dokument vorkommenden Worte - wenige Worte beschränkt, die auch nicht unbedingt in ihrer ursprünglichen, im Dokument enthaltenen, Form als Feature im Vektor[7] auftauchen. So bleibt weder der syntaktische noch der semantische Zusammenhang des textuellen Inhalts erhalten.

Das Erhalten des syntaktischen und semantischen Zusammenhangs der textuellen Inhalte wird auch durch Multi-Word-basierte Verfahren nur unwesentlich verbessert im Vergleich zu einzelwortbasierten Verfahren. Die Multi-Word-basierten Verfahren basieren darauf, nicht nur Einzelworte als Features zu verwenden, sondern eine

1 Eine genauere Erläuterung wird in einem späteren Kapitel dieser Arbeit erfolgen, siehe Kapitel 2.1.

2 Vgl. Sebastiani (2002), S. 10; Weiss u.a. (2005), S. 20; Manning u.a. (2008), S. 21 f.; Frigui u.a. (2004), S. 46; Dhillon u.a. (2004), S. 73; Solka (2008), S. 96; Aggarwal u.a. (2012), S. 167.

3 In der vorliegenden Arbeit werden „Einzelworte“, „einzelne Worte“ und „Worte“ synonym verwendet.

4 Vgl. Lewis u.a. (2004), S. 387; Sebastiani (2002), S. 10 f.

5 Darunter werden Worte verstanden, die sehr häufig in natürlichsprachlichen Texten verwendet werden und die deshalb wenig zur Bedeutung des Textes beitragen, vgl. Manning u.a. (2008), S. 25 f. Manning u.a. (2008), S. 26, schreiben jedoch auch, dass es in Information-Retrieval-Systemen zunehmend unüblich wird, überhaupt Stoppworte auszusortieren.

6 Unter Stemming versteht man das heuristische Entfernen von Wortenden, um diese auf eine Art „Stammform“ zurückzuführen, die bspw. bei einer Suche mehr Ergebnisse zurückliefern würde als die grammatikalisch angepassten Formen, vgl. Manning u.a. (2008), S. 30 f. Das bedeutet auf die Feature-Vektor-Erstellung übertragen, dass in Dokumenten vorkommende Worte auf eine Art „Stammform“ zurückgeführt werden, die in mehr Dokumenten auftaucht als die an den jeweiligen Inhalt angepasste Form des Wortes. Ein Beispiel wären hier die Worte „spieler“, „spiel“ und „spielerisch“, die in drei verschiedenen Dokumenten vorkommen könnten. Würden die Worte in ihrer eigentlichen Form als Features verwendet, so würden sie jeweils in dem entsprechenden Feature-Vektor des jeweiligen Dokuments auftauchen, jedoch nicht in denen der anderen beiden Dokumente. Führt man die drei Worte mit einem Verfahren auf ihren „Stamm“ zurück, so ergäbe sich bspw. „spiel“, das wiederum in allen drei Dokumenten auftaucht.

7 Die Begriffe „Vektor“ und „Feature-Vektor des Dokuments“ werden synonym verwendet.

Kombination von Worten. Die Kombinationen beruhen dabei auf grammatikalischen Strukturen[1] oder auf Distanzen zwischen den Worten, die als Multi-Word-Features ermittelt werden[2]. Die Probleme, die sich bei der Verwendung von Multi-Word-Features ergeben, sind:

- Verlust der Reihenfolge- und Distanzinformationen

  Genau so, wie es bei den einzelwortbasierten Features der Fall ist, werden auch bei den Multi-Word-Features nicht unbedingt aufeinanderfolgende Worte als Features ausgewählt. Stattdessen wird versucht, wahrscheinliche Wortkombinationen zu erzeugen, die aber nicht zwangsweise im Zusammenhang im ursprünglichen Text stehen. Dabei geht ebenfalls der syntaktische und semantische Zusammenhang verloren.

- Kombinatorische Explosion

  Wenn die Wortkombinationen nicht direkt aus dem Text selbst abgeleitet werden, sondern stattdessen durch Wahrscheinlichkeits- oder Distanzberechnungen künstlich erzeugt werden, entstehen zum einen insgesamt sehr viele Kombinationen und zum anderen viele Kombinationen, die nicht im Text vorkommen können.[3]

1 Dabei handelt es sich um so genannte n-Gramme. Insbesondere Bigramme, diese bestehen aus zwei Worten, und Trigramme, diese bestehen aus drei Worten, werden in der Klassifizierung verwendet. Um diese Bi- oder Trigramme zu finden, können grammatikalische Konstrukte verwendet werden, wie beispielsweise die Extraktion aller Bi- und Trigramme, bestehend aus „Adjektiv Nomen“ und „Adjektiv Nomen Nomen“ und deren Kombinationen, vgl. Johnson u.a. (2006), S. 4 (Seitennummerierung der Verfasserin, da Webdokument). Das wird als „part-of-speech“-Filterung bezeichnet. Zusätzlich können auch komplexere natürlichsprachliche Methoden zur Erzeugung der Bi- oder Trigramme angewendet werden, vgl. Papka u.a. (1998), S. 125. Teilweise werden aber auch die Texte einfach in aufeinanderfolgende Wortsequenzen zerlegt, vgl. bspw. Norvig (2009), S. 219. Andere Verfahren versuchen über Wahrscheinlichkeitsberechnungen Worte zu finden, die mit einer hohen Wahrscheinlichkeit in gleichen Sätzen auftauchen, vgl. Brown u.a. (1990), S. 80.

2 Bei diesen Verfahren geht es darum, Features zu ermitteln, die innerhalb des Textes eine bestimmte Anzahl an Worten auseinanderliegen und diese als Multi-Word-Features festzulegen. In den meisten Fällen wird versucht, Wortkombinationen zu finden, die nahe beieinander liegen, vgl. Hawking u.a. (1995), S. 134. Es gibt jedoch auch Ansätze, die das genaue Gegenteil verfolgen, vgl. Papka u.a. (1998), S. 125.

3 Vgl. Johnson u.a. (2006), S. 3 (Seitennummerierung der Verfasserin, da Webdokument). Die Autoren nennen hier ein Beispiel von 25.000 Worten, aus denen alle möglichen Bi- und Trigramme gebildet werden. Das bedeutet, es können $\frac{25.000!}{(25.000-2)!}$ unterschiedliche Bigramme erzeugt werden. Das entspricht fast 625 Millionen. Im Fall der Trigramme wären es über 15 Trillionen. In den Bigrammen oder Trigrammen sind also sehr viele Kombinationen enthalten, die nicht im Text auftauchen können.

- Gespaltene Meinung in der Literatur

  Es hielt und hält sich die Meinung, dass die Verwendung von Phrasen[1] eine bessere Textrepräsentation ermöglichen sollte. Croft u.a. (1991)[2] führen aus, dass diese Meinung durch die von ihnen zusammengetragenen Ergebnisse über die Qualität der auf den Multi-Word-Features basierenden Klassifizierungen oder Suchergebnissen aus früher verfassten Artikeln nicht bestätigt werden konnte. Sie berichten von gemischten Ergebnissen mit kleineren Verbesserungen, aber auch Verschlechterungen. Ihr eigener Ansatz, die Verwendung von Phrasen in Anfragen für das Information Retrieval, lässt sie aber Verbesserungen in diesem Bereich feststellen.[3]

  Strzalkowski (1996)[4] kommt zu dem Schluss, dass die Verwendung von Phrasen eine Verbesserung in dem dort beschriebenen Szenario bewirkt. Sebastiani (2002)[5] fasst zusammen, dass bisher noch keine wirklichen Verbesserungen bei der Verwendung von Phrasen beobachtet wurden, jedoch hieran noch weitergearbeitet wird und evtl. noch Verbesserungen entwickelt werden.

Zhang u.a. (2007)[6] verfolgen noch einen anderen Ansatz. Sie erzeugen Multi-Word-Features, jedoch nicht mittels der oben beschriebenen Verfahren, sondern durch den direkten Vergleich jeweils zweier Sätze der entsprechenden Texte einer Klasse. Zwar werden dann ebenfalls Features erzeugt, die wirklich im Text auftreten, jedoch keine wortübergreifenden Features[7]. Zusätzlich wird die Festlegung, ob eines dieser Features in den Dokumenten auftaucht, über einen Prozentanteil der enthaltenen Einzelworte oder eine Distanz zwischen den enthaltenen Einzelworten geregelt.[8] Das bedeutet in diesem Fall, dass wieder vom eigentlichen textuellen Inhalt des Dokuments abgewichen wird.

1.4 Wissenschaftliche Problemstellung

Um eine erhöhte Qualität ohne Zusatzwissen bei der Aufbereitung von textuellen Daten zu erreichen, wird in der vorliegenden Arbeit Abstand von der einzelwort- oder

1 In der Literatur werden unter dem Begriff „Phrase“ die oben beschriebenen n-Gramme und Multi-Word-Features, die über Distanzinformationen gewonnen werden, zusammengefasst. Im Hinblick auf die n-Gramme spielt es keine Rolle, mit welchem Verfahren sie gewonnen werden.
2 Vgl. Croft u.a. (1991), S. 32.
3 Vgl. Croft u.a. (1991), S. 42.
4 Vgl. Strzalkowski (1996), S. 147.
5 Vgl. Sebastiani (2002), S. 10 f.
6 Vgl Zhang u.a. (2007), S. 3520.
7 Mit *wortübergreifenden Features* sind jetzt und im Folgenden Features gemeint, die nicht zwangsläufig aus einem oder mehreren vollständigen Worten bestehen müssen, aber können.
8 Vgl. Zhang u.a. (2007), S. 3521.

Multi-Word-basierten Aufbereitung der textuellen Daten genommen. Stattdessen wird eine Aufbereitung von textuellen Daten mit einer *wortübergreifenden* Basis untersucht. Wortübergreifende Features definieren sich in der vorliegenden Arbeit wie folgt:

Definition 1. *Ein **wortübergreifendes** Feature besteht aus einer Sequenz von Zeichen aus einem natürlichsprachlichen Text. Diese Zeichensequenz wird nicht durch Wortgrenzen wie Leer- oder Satzzeichen begrenzt. Das bedeutet, wortübergreifende Features können Einzelworte, Wortsequenzen, aber auch Fragmente von Einzelworten oder Wortsequenzen sein.*

Die wortübergreifenden Features sollen ohne Zusatzwissen in einem ersten Schritt automatisch erzeugt werden. Zusatzwissen meint in der vorliegenden Arbeit sowohl Wissen über die Daten an sich als auch technisches Wissen. Durch die „Zerstückelung“ des eigentlichen Textes eines Dokuments gehen Informationen verloren. Das gilt für alle vorgestellten Vorverarbeitungsarten, auch für die Erzeugung von wortübergreifenden Features. Es existiert immer eine Auswahl, welche Features verwendet werden, um das Dokument zu repräsentieren. Das beinhaltet auch immer ein „Weglassen“ anderer Textteile, die jedoch in bestimmten Situationen ebenfalls einen Beitrag zur Qualitätssteigerung von automatischen Klassifizierungen und Cluster-Vorgängen leisten könnten.
Anstatt, wie für die wortbasierten Features beschrieben, ein Stemming und ein Weglassen von Stoppworten durchzuführen, wird dies für die wortübergreifenden Features nicht durchgeführt, um nötiges Zusatzwissen zu begrenzen.[1]
Es besteht eine Wissenslücke im Hinblick auf den Einsatz von wortübergreifenden Features als Features innerhalb der Klassifizierung und des Clusterns, da sie bisher nicht als Features zum Klassifizieren und Clustern eingesetzt werden.[2]

1 Das „Nachbessern“ der Ergebnisse einer Klassifizierung und eines Clusterns durch menschliche Experten sollte jedoch möglich bleiben, da eine vollständige Automatisierung der Klassifizierung und des Clusterns, die 100% aller textuellen Daten richtig klassifiziert, ohne Einfließen von Zusatzwissen wahrscheinlich nicht durchführbar ist, vgl. Spangler u.a. (2007), S. 9.

2 Die Literatur erwähnt nach Kenntnis der Verfasserin wortübergreifende Features, wie sie hier definiert sind, für das Klassifizieren und Clustern nicht.

Daraus resultiert das wissenschaftliche Problem der Arbeit:

> **Das wissenschaftliche Problem der vorliegenden Arbeit besteht darin zu überprüfen, ob wortübergreifende Features im Vergleich zu einzelwortbasierten Features qualitativ[1] bessere Ergebnisse bei der Klassifizierung oder dem Clustern von natürlichsprachlichen Texten liefern, ohne dass in einem ersten Schritt explizites Zusatzwissen benötigt wird.**

Ergäbe eine Überprüfung qualitativ bessere Ergebnisse, wäre eine eventuelle Lösung des Realproblems der noch nicht ausreichenden Qualität der durchgeführten Klassifizierungen oder des Clusterns von natürlichsprachlichen Texten ohne Zusatzwissen für das Ermitteln der wortübergreifenden Features gefunden.
Das wissenschaftliche Problem ist nicht-trivial, da die relevanten technischen Herausforderungen im Zusammenhang mit Klassifizierung und Clustern Hürden darstellen. Zu nennen sind unter anderem der Umgang mit großen Datenmengen, die inhärente Problemschwere[2], die hohe Dimensionalität, die Spärlichkeit der Featureverteilungen in Dokumenten, die häufige Nicht-Normalverteilung der Features etc.[3]

1.5 Arbeitstechniken

Die Aufbereitung der natürlichsprachlichen Texte soll mit wortübergreifenden Features erfolgen. In Anlehnung an Gusfield (1999)[4] ist ein Text eine geordnete Abfolge von Zeichen. Für jeden Text T heißt $T[i..j]$ der Teiltext von T, der an der Position i beginnt und an j endet, wobei $1 \leq i, j \leq |T|$ und $i \leq j$. $|T|$ bezeichnet die Länge des Textes T und $T[i..|T|]$ ist das Suffix des Textes T, das an Position i beginnt. Ein Suffix des Wortes „Beispiel“ ist „spiel“, welches an Position 4 beginnt und am Ende des Wortes endet. Bei einem Text-Suffix-Fragment handelt es sich um einen Teil eines Suffixes des Textes.[5]
Die in der vorliegenden Arbeit vorgeschlagene Vorverarbeitung identifiziert alle Text-Suffix-Fragmente, die in einem Dokument oder der Klasse des Dokuments mehrfach

1 Qualitätsmaße in Form von Evaluationsmaßen zur Bestimmung der Qualität der Ergebnisse des Klassifizierens und des Clusterns werden in Kapitel 2.5 dieser Arbeit eingeführt.
2 Clustern ist in seiner allgemeinen Form ein NP-hartes Problem, vgl. Strehl (2002), S. 8.
3 Vgl. Strehl (2002), S. 11 f.
4 Gusfield (1999), S. 3 f.
5 Eine genaue Definition von Text-Suffix-Fragment und den daraus erzeugten Features befindet sich in Kapitel 3.2.1 der vorliegenden Arbeit.

vorkommen und eine Mindestlänge von drei Zeichen haben, und legt diese als Features für das jeweilige Dokument oder für die jeweilige Klasse fest.[1]
Durch die Verwendung der Datenstruktur Suffix Tree oder Suffix Array[2] zur Aufbereitung der Dokumente und zur Ermittlung der Features werden Text-Suffix-Fragmente als Features ermittelt, die Einzelworte, Fragmente von Einzelworten, Fragmente von Wortkombinationen und auch Wortkombinationen enthalten. Das Entscheidende bei dieser Art der Aufbereitung ist, dass durch die Aufbereitung als Textfragmente im Gegensatz zu wortbasierten Verfahren auch wortübergreifende Features erzeugt werden können, die aber, im Gegensatz zu Multi-Word-Features, nicht aus einem Wort oder mehreren vollständigen Worten bestehen müssen.
Mit dieser Art der Aufbereitung werden die syntaktischen Zusammenhänge in den verarbeiteten Daten bewahrt.[3] Dies führt im Vergleich zu Einzelwortverfahren zunächst zu einem Anwachsen der Featuremenge, schränkt die Featurezahl im Vergleich zu Multi-Word-Verfahren aber gleichzeitig in natürlicher, sequenzorientierter Weise ein. Die Strukturen, die zur Verarbeitung dieser erweiterten Informationsbasis verwendet werden sollen - der Suffix Tree oder das Suffix Array - versprechen somit verbesserte Möglichkeiten, das Clustern und die Klassifizierung zu unterstützen.
Die vorgeschlagenen Datenstrukturen wurden bisher hauptsächlich zur Verarbeitung von DNA-Sequenzen[4] oder innerhalb der Kompression[5] eingesetzt. Auch die Verwendung von Suffix Trees innerhalb des Clusterns[6] beruht auf der Verarbeitung von ganzen Worten, d.h., es werden keine wortübergreifenden Features verwendet.
Für die Bewertung der erreichten Qualität von Klassifizierung und Clustern wird die folgende Evaluation durchgeführt: Benchmark mit dem Reuters-Testdatensatz[7].
Motivation: Dieser Testdatensatz wurde, ebenso wie sein Vorgänger, in zahlreichen Studien zur Bewertung der Leistungsfähigkeit von Klassifizierungsverfahren

1 Diese Parameter wurden von der Verfasserin gewählt. Die Motivation dieser Wahl wird in einem späteren Kapitel dieser Arbeit erläutert, siehe Kapitel 3.2.1.
2 Weitere Erläuterungen und Definitionen befinden sich in Kapitel 3.1.
3 Zusätzlich können die zuvor genannten grammatik- oder distanzbasierten Verfahren auch im Falle einer auf Text-Suffix-Fragmenten basierenden Vorverarbeitung zum Einsatz kommen, da die relevanten Einzelworte auch enthalten sein können. Zudem bietet sich die Möglichkeit, Text-Suffix-Fragment-basierte Grammatiken zu entwickeln, die die zusätzlichen Informationen, die die neue Vorverarbeitung im Vergleich zu wortbasierten Verfahren liefert, verwenden. Das ist aber nicht Gegenstand der vorliegenden Arbeit.
4 Vgl. Gusfield (1999). Bei der Verarbeitung von DNA-Sequenzen ist ein Vorteil, dass das zu Grunde liegende Alphabet oder Vokabular nur 4 Zeichen - „G“, „A“, „T“ und „C“ - umfasst. Werden, wie in der vorliegenden Arbeit, natürlichsprachliche Texte verarbeitet, so wächst die Anzahl der zu verarbeitenden Zeichen um ein Vielfaches an und die Komplexität wird erhöht. Diese Komplexität existiert im Bereich von DNA-Sequenzen nur in einigen wenigen Fällen, vgl. Gusfield (1999), S. 155 f.
5 Vgl. Adjeroh u.a. (2008), S. 51.
6 Vgl. Zamir u.a. (1998), S. 48.
7 Vgl. NIST (2004).

herangezogen.[1] Er bietet das Potenzial, die Leistungsfähigkeit der neuartigen Vorverarbeitung unter Verwendung typischer Klassifizierungs- und Clusterverfahren zu demonstrieren.

Vorgehensweise: Es wird ein Vergleich zwischen der neuen Vorverarbeitung und der Verarbeitung der Dokumente mit einzelwortbasierten Features vorgenommen.[2] Hierzu werden die durch D.D. Lewis[3] zur Verfügung gestellten Vektoren, die auf Einzelworten basieren, benutzt, um mit Hilfe der Trainingsdaten und Algorithmen die Testdaten zu klassifizieren und zu clustern. Anschließend wird die Qualität der durchgeführten Klassifizierung und des Clusterns gemessen.[4] Der gleiche Ablauf erfolgt für die gleichen Trainings- und Testdaten und Algorithmen mit Hilfe der in dieser Arbeit vorgeschlagenen neuen Vorverarbeitung zur Erzeugung der Features. Abschließend werden die erreichten Qualitätsergebnisse einander gegenübergestellt.

1.6 Intendierte wissenschaftliche Ergebnisse der Arbeit

Als wissenschaftliche Ergebnisse werden folgende Erkenntnisse angestrebt:

- eine Definition für wortübergreifende Features als Text-Suffix-Fragment-Features für natürlichsprachliche Texte,
- ein Algorithmus zur automatischen Erzeugung dieser Features,
- ein Software-Protoyp, der die automatische Erzeugung der Text-Suffix-Fragment-Features für natürlichsprachliche Texte vornimmt und die Klassifizierung von Texten sowie das Clustern von Texten durchführt, sowie
- eine Empfehlung der Verwendung von Text-Suffix-Fragment-Features für natürlichsprachliche Texte für das Klassifizieren und das Clustern. Diese Empfehlung basiert auf einem Vergleich der Qualität der erzeugten Ergebnisse beim Einsatz der wortübergreifenden Features und der Qualität der erzeugten Ergebnisse beim Einsatz von einzelwortbasierten Features.

1 Vgl. Lewis u.a. (2004), S. 393; Manning u.a. (2008), S. 142.
2 Von einem Vergleich mit Multi-Word-Features wird abgesehen, da im Hinblick auf die Erzeugung der Multi-Word-Features eine Vielzahl an Möglichkeiten zur Verfügung steht, die die Ergebnisse beeinflussen könnten. Für den Reuters-Testdatensatz werden einzelwortbasierte Features zur Verfügung gestellt, die für den Vergleich verwendet werden können.
3 Vgl. Lewis (2004), Appendix 13.
4 Wie genau diese Messungen erfolgen wird in Kapitel 2.5 erläutert.

1.7 Aufbau der Arbeit

Nach dem einleitenden Kapitel werden in Kapitel 2 die Grundlagen von Klassifizierung und Clustern von natürlichsprachlichen Texten dargestellt. Dazu gehören eine Erläuterung der Vorverarbeitung der Texte, die Vorstellung von Ähnlichkeits- und Distanzmaßen, die Definition der Klassifizierung mit einer Erläuterung ausgewählter Algorithmen, die Definition des Clusterns mit einer Erläuterung ausgewählter Algorithmen und schließlich die Erläuterung der Evaluationsmaße für das Klassifizieren und Clustern natürlichsprachlicher Texte.
Kapitel 3 stellt die zentralen Datenstrukturen - Suffix Tree und Suffix Array - vor, die zur Erzeugung der wortübergreifenden Features verwendet werden. Zusätzlich wird der Algorithmus zur Erzeugung von Suffix Arrays, der in der vorliegenden Arbeit verwendet wird, erläutert, und darauf aufbauend der Algorithmus zur Erzeugung von generalisierten Suffix Arrays, der entscheidend für die Ermittlung der Text-Suffix-Fragement-Features ist. Zusätzlich zur Erläuterung der Algorithmen erfolgt eine Beschreibung der Implementierung dieser Datenstrukturen und Algorithmen im Prototyp. Abschließend werden in Kapitel 3 die in der Arbeit verwendete Definition der Text-Suffix-Fragment-Features ausgeführt, der Algorithmus zur Ermittlung dieser Features vorgestellt und die Implementierung dieser Feature-Ermittlung im Prototyp beschrieben.
Kapitel 4 enthält die Beschreibung des Klassifizierens von natürlichsprachlichen Texten. Es wird die Implementierung des Klassifizierens im Prototyp sowohl für die wortübergreifenden Features als auch für die einzelwortbasierten[1] Features dargestellt. Der Benchmark mit dem Reuters-Testdatensatz wird in Form von Experimenten mit verschiedenen Klassifizierungsalgorithmen und Evaluationsmaßen beschrieben und die Ergebnisse der wortübergreifenden und der wortbasierten Features werden einander vergleichend gegenübergestellt.
Kapitel 5 enthält eine Darstellung des Clusterns von natürlichsprachlichen Texten. Im Kapitel enthalten ist ebenfalls eine Erläuterung der Implementierung des Clusterns im Prototyp, sowohl für die wortübergreifenden als auch für die wortbasierten Features. Die Benchmark-Ergebnisse für verschiedene Cluster-Algorithmen, Distanz- und Ähnlichkeitsmaße und Evaluationsmaße sind in Form von Experimenten dargestellt. Die Darstellung enthält ebenfalls einen Vergleich der von den beiden Verfahren zur Feature-Erzeugung erreichten Ergebnisse.

1 In der vorliegenden Arbeit werden im Folgenden „einzelwortbasierte Features“ und „wortbasierte Features“ synonym verwendet.

Kapitel 6 enthält das Fazit der Verfasserin und eine Empfehlung für die Verwendung von Text-Suffix-Fragmenten als Features im Rahmen der Klassifizierung und des Clusterns natürlichsprachlicher Texte.
In Kapitel 7 wird in einem Ausblick auf weitere relevante Forschungsthemen eingegegangen.
Der Anhang enthält unter anderem eine Kurzbeschreibung des Inhalts der beiliegenden CD. Darauf befindet sich die Implementierung des Software-Prototyps.

2 Grundlagen

2.1 Repräsentation von Texten

2.1.1 Definition grundlegender Begriffe

2.1.1.1 Dokument

Bei einem Dokument d_i handelt es sich laut Feldman u.a. (2007), S. 3, um zusammenhängende textuelle Daten innerhalb einer Dokumentsammlung, die mit Dokumenten in der realen Welt korrelieren können. Manning u.a. (2008) definieren ein Dokument genereller, d.h., als eine wie auch immer geartete Einheit, für die ein Information Retrieval System entwickelt wird.[1]

In der vorliegenden Arbeit ist ein Dokument ein zusammenhängender Text. Für die durchgeführten Experimente gilt außerdem, dass der Text elektronisch vorliegt und Teil einer in den Experimenten verwendeten Dokumentsammlung ist. Es zeichnet sich durch eine eigene Identifikation aus, d.h., es besitzt eine Nummer, die es eindeutig kennzeichnet. Des Weiteren kann das Dokument beliebig aufgebaut sein. Es existiert also keine Vorschrift, wie es strukturiert ist.[2]

2.1.1.2 Dokumentsammlung

Eine Dokumentsammlung D bezeichnet eine Menge an Dokumenten. Eine andere Bezeichnung für eine Dokumentsammlung ist *corpus*.[3] Der Zweck der Zusammenstellung einer Dokumentsammlung ist bspw. das Testen der Qualität von Klassifizierern oder das Auffinden von Zusammenhängen zwischen diesen Dokumenten und dem Ableiten von Informationen und Wissen zum Lösen von Problemstellungen.

In der vorliegenden Arbeit werden Dokumentsammlungen verwendet, um die Qualität von neuartigen Dokumentrepräsentationen im Zusammenhang mit dem Klassifizieren und dem Clustern dieser Dokumente untersuchen zu können.

1 Vgl. Manning u.a. (2008), S. 4. Für eine Definition von Information Retrieval vgl. Manning u.a. (2008), S. 1.

2 Zur Unterscheidung zwischen strukturierten und unstrukturierten Daten vgl. Kapitel 1.1 auf S. 1 dieser Arbeit. Des Weiteren können bspw. diesbezüglich auch Weiss u.a. (2005), S. 2-6, und Feldman u.a. (2007), S. 3 f., herangezogen werden.

3 Vgl. Manning u.a. (2008), S. 4.

2.1.1.3 Dokumentrepräsentation

Um Dokumente klassifizieren oder clustern zu können, müssen sie von einer Form, in der sie für den Computer lediglich eine Aneinanderreihung von Zeichen darstellen, die nicht durch ein Datenmodell festgelegt ist, in eine Form gebracht werden, die vom Computer verarbeitet werden kann. Sie müssen in eine Form umgewandelt werden, die von den verschiedenen Algorithmen zum Klassifizieren und zum Clustern auch genutzt werden kann.[1] Die Dokumente müssen auf eine bestimmte Art und Weise repräsentiert werden.
Das geschieht laut Sebastiani (2002)[2], indem eine Operation auf die einzelnen Dokumente angewendet wird, die diese indiziert. Das bedeutet, das Dokument wird in eine kompakte Form gemappt. Diese besteht in den meisten Fällen aus einem Vektor von Gewichten, die sich auf so genannte Terme (engl. *terms*) oder Features des Dokuments beziehen.[3] In der vorliegenden Arbeit wird ein Dokument d_i durch einen Feature-Vektor $\vec{f}_{d_i}$ mit den Komponenten $\vec{f}_{d_i} = (f_{1_{d_i}}, f_{2_{d_i}}, ..., f_{|F|_{d_i}})$ repräsentiert, wobei $|F|$ die Gesamtanzahl der Features in der Featuremenge F der Dokumentsammlung D meint.

2.1.2 Erstellung einer Dokumentrepräsentation

2.1.2.1 Überblick

Der Ablauf der Erstellung einer Dokumentrepräsentation ist bildlich in Abbildung 2.1 auf S. 21 zu sehen.[4] Dieser Ablauf ist im Großen und Ganzen Weiss u.a. (2005)[5] sowie Manning u.a. (2008)[6] entnommen, wurde jedoch verfeinert und ergänzt.

2.1.2.2 Dokumentsammlung erstellen

Der erste Schritt auf dem Weg zu einer Dokumentrepräsentation ist das Sammeln von Dokumenten. Die Sammlung der Dokumente kann einerseits auf die Problemstellung, die es zu bearbeiten gilt, ausgerichtet sein. Andererseits kann aber auch der Fall eintreten, dass Dokumente gesammelt wurden, die nun weiterverarbeitet werden sollen. Als Beispiel für eine solche Sammlung kann das Beispiel aus dem Einleitungskapitel dieser Arbeit dienen: Hier wurden E-Mails „gesammelt“, die wei-

1 Vgl. bspw. Sebastiani (2002), S. 10.
2 Vgl. Sebastiani (2002), S. 10.
3 Im nachfolgenden Kapitel 2.1.2 erfolgen weitere Erläuterungen zu Termen und Features.
4 Die Erläuterung des Ablaufs einschließlich aller Begriffe befindet sich in den nachfolgenden Kapiteln.
5 Vgl. Weiss u.a. (2005), S. 15-46.
6 Vgl. Manning u.a. (2008), S. 6, 18, 21-33.

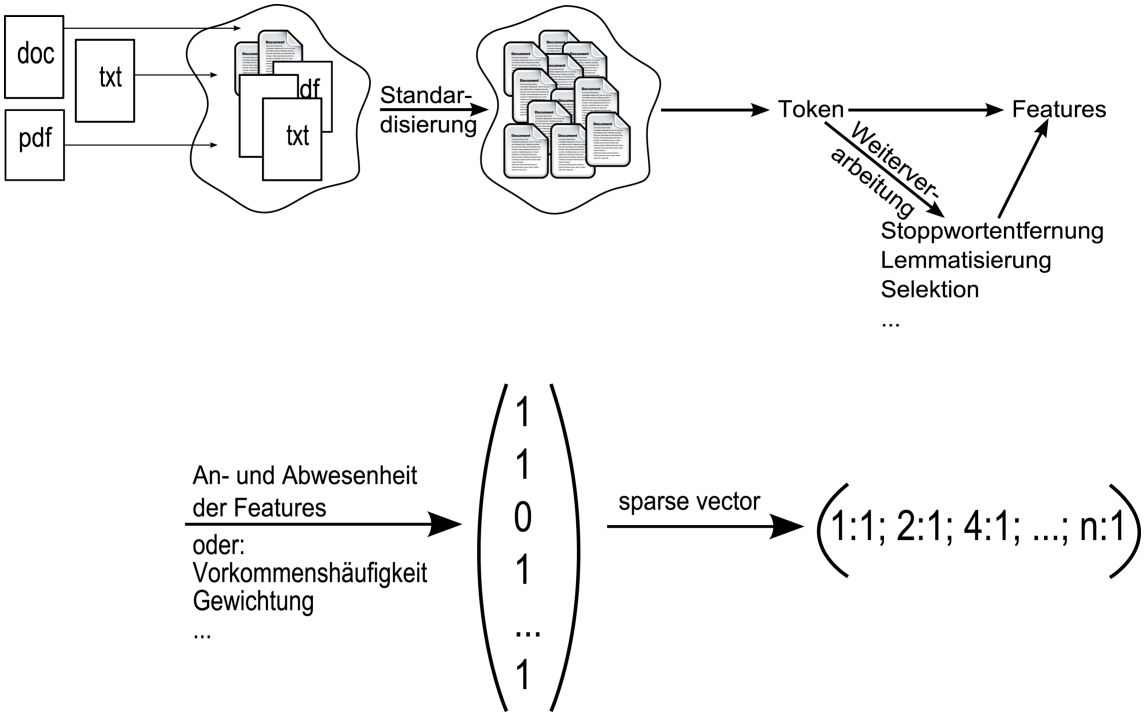


Abbildung 2.1: Ablauf des Erstellens einer Dokumentrepräsentation

terverarbeitet werden müssen. In der vorliegenden Arbeit entfällt dieser Schritt, da bereits verfügbare Dokumentsammlungen in den Experimenten verwendet werden.

2.1.2.3 Standardisierung von Dokumenten der Dokumentsammlung

Wird eine neue Dokumentsammlung erstellt, kann es vorkommen, dass die einzelnen Dokumente nicht im gleichen Format vorliegen. Das bedeutet zum Beispiel, dass die Dokumente als PDF-Datei, Word-Datei, ASCII-Datei oder in anderen Formaten gemischt in der Sammlung vorliegen. In diesen Fällen müssen die Dokumente alle in das gleiche Format gebracht werden. Das ist in den meisten Fällen das XML-Format[1].
In der vorliegenden Arbeit entfällt dieser Schritt ebenfalls, da die Dokumente der verwendeten Dokumentsammlungen bezogen auf ihr Format bereits standardisiert sind.

2.1.2.4 Features erzeugen

Features dienen dazu, das Dokument, zu dem sie gehören, möglichst gut darzustellen, so dass es so repräsentiert wird, dass Algorithmen es weiterverarbeiten können. Dafür

1 Für eine Erläuterung siehe Fußnote 1 auf S. 243 der vorliegenden Arbeit.

werden aus dem ursprünglich zusammenhängenden Text diese Features erzeugt. Der Text wird also in kleinere Einheiten zerlegt, die in ihrer Gesamtheit das Dokument repräsentieren.[1] Für diese Zerlegung existieren unterschiedliche Ansätze, je nachdem, was die Features eines Dokuments sein sollen. Mögliche Features sind[2]:

- Zeichen
  Jedes Dokument ist ein zusammenhängender Text. Dieser Text besteht aus aneinandergereihten Zeichen, in die er zerlegt werden kann. Bei einer solchen Zerlegung des Textes bilden die Zeichen, die darin auftauchen, seine Features und somit die Features des Dokuments.

- Worte
  Ein Dokument lässt sich auch in größere Einheiten als Zeichen zerlegen, wie zum Beispiel in Worte. Dieser Ansatz ist der gebräuchlichste beim Klassifizieren und Clustern von natürlichsprachlichen Texten.[3]

- Terme
  Verwendet man nicht nur einzelne Worte, sondern auch Multi-Words, also mehr als ein Wort[4] als Feature, so bezeichnet man solche Features als Terme.[5] Um diese Art von Features zu finden, existieren spezielle Verfahren, da sie nicht einfach durch eine Zerlegung des Textes gefunden werden können.[6]

- Konzepte
  Schließlich existieren noch Konzepte als Features für ein Dokument. Der Unterschied zu allen anderen Featurearten besteht darin, dass hier Features für ein Dokument gefunden werden, die so nicht im eigentlichen Text des Dokuments auftauchen. Konzepte bestehen aus Worten oder Multi-Words aus dem Text, aber auch zusätzlichen Worten oder Multi-Words, die durch Zusatzinformationen von außerhalb des Dokuments der Featuremenge des Dokuments hinzugefügt werden.

1 Vgl. Feldman u.a. (2007), S. 4.
2 Vgl. Feldman u.a. (2007), S. 5-7.
3 Vgl. Sebastiani (2002), S. 10; Weiss u.a. (2005), S. 20; Manning u.a. (2008), S. 21 f.; Frigui u.a. (2004), S. 46; Dhillon u.a. (2004), S. 73; Solka (2008), S. 96; Aggarwal u.a. (2012), S. 167.
4 Für eine Erläuterung siehe S. 10 dieser Arbeit.
5 In der vorliegenden Arbeit wird *Features* als Bezeichnung für die letztendlich darzustellenden Repräsentanten eines Textes gewählt. Andere Autoren, wie z.B. Manning u.a. (2008), S. 3, bezeichnen die Repräsentanten eines Textes als *Terme*.
6 Diese Verfahren bezeichnet man als Termextraktion (engl. *term extraction*). Beispiele für Verfahren zur term extraction befinden sich in Feldman u.a. (1998), S. 9-2, 9-4 f.; Piao u.a. (2010), S. 1 f. (Seitennummerierung der Verfasserin); Bosma u.a. (2010), S. 2278-2280; Bonin u.a. (2010), S. 3222-3225.

Je nachdem, welche Art von Features für ein Dokument verwendet werden soll, erfolgt eine unterschiedliche Verarbeitung des Dokuments um diese Features zu finden. In der vorliegenden Arbeit wird eine neue Art von Features eines Dokuments vorgestellt, die mit dem wortbasierten Ansatz verglichen werden soll. Aus diesem Grund wird an dieser Stelle nur das Vorgehen zum Finden von wortbasierten Features für ein Dokument beschrieben, jedoch nicht das Vorgehen zum Finden von Termen oder Konzepten.[1]

Um wortbasierte Features eines Dokuments zu finden, muss das Dokument in die in ihm auftauchenden Worte zerlegt werden. Weiss u.a. (2005) bezeichnen diesen Vorgang als Tokenisierung (engl. *tokenization*).[2] Damit meinen sie jedoch generell die Zerlegung des Dokuments in so genannte *Token*, die durchaus auch Zeichen oder Terme sein können. An dieser Stelle wird aber nur die Zerlegung in Worte betrachtet. Die Menge der Token bildet die Menge der *potenziellen* Features F'_{d_i} des Dokuments d_i. Potenziell ist diese Menge deshalb, da es nach der Zerlegung des Dokuments in Token zwei Möglichkeiten gibt weiterzuverfahren:

1. Die Token werden nicht weiterverabeitet und bilden die Menge der Features des Dokuments[3], also die Auswertung von $F'_{d_i} == F_{d_i}$ ergibt „wahr“.

2. Die Token werden weiterverarbeitet, daraus werden Features abgeleitet und so wird die Menge F_{d_i} der Features des Dokuments gebildet.

Im ersten Fall erfolgt keine Weiterverarbeitung der Token. Sie bilden nicht nur die Menge der potenziellen Features, sondern auch die Menge der Features des Dokuments.

Im zweiten Fall werden dagegen die gefundenen Token weiterverarbeitet. Die Weiterverarbeitung besteht in den meisten Fällen aus einer Selektion von Features aus der Menge der Token und einer Verallgemeinerung der selektierten Token. Folgende Weiterverarbeitungen sind beispielsweise möglich[4]:

- Stoppwortentfernung
  Aus der Menge der Token können solche Token entfernt werden, die nichts zur Bedeutung beitragen, so genannte Stoppworte.[5]

1 Neben den in Fußnote 6 auf S. 22 genannten Beispielen zur term extraction finden sich Beispiele für das Extrahieren von Konzepten in: Abebe u.a. (2010), S. 156 f.; Kumaran u.a. (2012), S. 45-46; Dinh u.a. (2011), S. 1161. Allgemein zur Informationsextraktion siehe bspw. Feldman u.a. (2002), S. 349-359.

2 Vgl. Weiss u.a. (2005), S. 20 f.

3 Die Menge der Features eines Dokuments wurde bereits als die Menge der zu verwendenden Features bezeichnet. Beide Bezeichnungen werden synonym verwendet.

4 Vgl. Weiss u.a. (2005), S. 21-43.

5 Vgl. auch Fußnote 5 auf S. 10.

- Lemmatisierung
  Unter Lemmatisierung wird bei Weiss u.a. (2005) das Stemming der Token verstanden.[1] Er trennt dabei zwischen dem grammatikalischen Stemming[2] und dem agressiven Stemming zu einer evtl. künstlichen Wurzel[3]. Manning u.a. (2008) unterscheiden dagegen Stemming und Lemmatisierung voneinander.[4] Sie verstehen unter Stemming das heuristische „Abschneiden“ von Wortteilen, also das, was Weiss u.a. (2005) als *stemming to a root* bezeichnen, und unter Lemmatisierung die Rückführung des Tokens auf seine grammatikalische Grundform, das *inflectional stemming* bei Weiss u.a. (2005). Insgesamt wird also bei der Lemmatisierung des Tokens dieses auf eine vorher festzulegende Art auf seine „Grundform“ reduziert, um mehr Übereinstimmungen zu finden.

- Selektion
  Eine Selektion dient dazu, die Menge der Features zu verkleinern. Auch hier existieren verschiedene Ansätze, das durchzuführen:

  – Selektion aufgrund der Vorkommenshäufigkeit
    Die Häufigkeit des Vorkommens eines Tokens in einem Dokument oder auch in der Dokumentsammlung wird gezählt. Legt man nun Schwellenwerte fest, die eine Mindest- und oder eine Maximalvorkommenshäufigkeit regeln, werden nur die Token als Features ausgewählt, die über oder unter den entsprechenden Schwellenwerten liegen. So wird eine Auswahl getroffen.

1 Vgl. Weiss u.a. (2005), S. 21. Vgl. auch Fußnote 6 auf S. 10.
2 Dabei handelt es sich um die Zurückführung von Worten auf ihren gemeinsamen Wortstamm, also bspw. „book“ und „books“ auf die gemeinsame Grundform „book“, vgl. Weiss u.a. (2005), S. 21. Im Englischen wird das *inflectional stemming* genannt.
3 Hierbei werden keine grammatikalischen Regeln berücksichtigt, sondern nur die syntaktischen Gemeinsamkeiten von Worten. So würde bspw. „denormalization“ zu „norm“ gestemmt, vgl. Weiss u.a. (2005), S. 23. Die Autoren nennen das *stemming to a root*.
4 Vgl. Manning u.a. (2008), S. 30 f.

  - Selektion aufgrund der Wichtigkeit
    Spezielle Methoden versuchen, anhand der gefundenen Token und der Dokumente wichtige Token aus der Menge der Token herauszufiltern. Wichtig heißt in diesem Zusammenhang, dass diese Token besonders gut dazu geeignet sind, Vorhersagen über die Dokumentsammlung zu tätigen, also bspw. besonders gut dazu geeignet sind, neue Dokumente zu klassifizieren. Dafür kann eine Reihenfolge der Wichtigkeit der Token für eine Klasse gebildet werden.[1]

- Linguistische Weiterverarbeitung
  Die Token können auch in linguistischer Hinsicht weiterverarbeitet werden und so zu Features des Dokuments werden. Dafür wird zumeist zunächst ein so genanntes *part-of-speech tagging* durchgeführt. Das bedeutet, jedes Token erhält seine sprachliche Klasse zugewiesen, wie bspw. Nomen, Verb, Adjektiv. Darauf aufbauend können weitere Analysen vorgenommen und Features ausgewählt oder erzeugt werden.

Ist eine oder sind mehrere dieser Weiterverarbeitungsarten abgeschlossen, so ist aus der Menge der Token die Menge der Features entstanden, die ein Dokument - und über alle Dokumente einer Dokumentsammlung - die Dokumentsammlung repräsentieren.

2.1.2.5 Vektoren erstellen

Verfügt man über alle Features F aller Dokumente in der Dokumentsammlung D, so nennt man diese Menge an Features *dictionary*.[2] Mit Hilfe der Gesamtmenge an Features F lassen sich Vektoren für alle Dokumente der Dokumentsammlung erstellen. Diese Vektoren bilden die letztendliche Repräsentation jedes Dokuments. Die Erstellung der Vektoren ist eine Überführung des Dokuments in eine numerische Form. Weiss u.a. (2005) beschreiben die Erstellung der Vektoren als die Erzeugung

1 Weiss u.a. (2005) benutzen dafür den Information Gain, vgl. Weiss u.a. (2005), S. 35 f.; zum Information Gain siehe Kapitel 2.3.3.3 dieser Arbeit. Lewis u.a. (2004) benutzen dagegen eine komplexere Berechnung: Das Auswahlkriterium ist eine durch die zuvor berechnete χ^2-Verteilung für jedes Token und für jede Klasse maximale Punktzahl für das betreffende Token. Zusätzlich muss diese Punktzahl auf der Rangliste noch innerhalb der festgelegten Anzahl an insgesamt auszuwählenden Features liegen. D.h. also, die maximale Punktzahl eines Tokens entspricht dem maximalen Wert der χ^2-Verteilung, berechnet über alle Klassen. Aufgrund dieser Werte wird eine Rangliste über alle Token erstellt, die sie vom besten bis zum schlechtesten aufführt. Anhand der vorher festgelegten Anzahl der Features wird die entsprechende Auswahl getroffen. Vgl. Lewis u.a. (2004), S. 386, 383.

2 Vgl. Weiss u.a. (2005), S. 25.

einer Tabelle, wobei jede Zeile ein Dokument und jede Spalte ein Feature darstellt.[1] Die Zeilen bilden jeweils den Vektor des entsprechenden Dokuments.
Es existieren unterschiedliche Arten, die Features mit Werten zu belegen:

- Binäre Werte
  Die einfachste Darstellung der Werte eines Vektors ist die binäre. Das bedeutet, der Vektor enthält nur die Informationen darüber, ob das entsprechende Feature im Dokument enthalten ist oder nicht. Dies wird durch Einsen und Nullen dargestellt.

- Häufigkeitswerte
  Soll nicht nur die An- oder Abwesenheit von Features in einem Dokument als Grundlage für die Weiterverarbeitung des Dokuments dienen, kann auch die Häufigkeit, mit der das entsprechende Feature im Dokument auftaucht, im Vektor gespeichert werden. Wenn Häufigkeiten angegeben werden, ergänzen sie die binären Informationen über die An- und Abwesenheit von Features im Dokument um die Zusatzinformation der Häufigkeit des Auftretens der Features. So können bei der Weiterverarbeitung Unterschiede zwischen Dokumenten stärker hervortreten.[2]

- Gewichtung
  Sollen nicht nur Häufigkeiten als Unterscheidungsmerkmal zwischen Dokumenten dienen, sondern die Wichtigkeit der Features, so können auch Werte in den Vektoren gespeichert werden, die aufgrund einer Gewichtung der Features entstehen. Dabei gibt es verschiedene Möglichkeiten der Gewichtung:

  – Position der Features
    Bei dieser Art der Gewichtung geht man davon aus, dass die, vom Menschen erfassbare, aber nicht durch Metainformationen ausgezeichnete, Struktur des Dokuments auch etwas über die Wichtigkeit der Worte innerhalb des Textes aussagt. So kann man bspw. davon ausgehen, dass Worte aus einer Überschrift den Inhalt des Dokuments besser repräsentieren als Worte aus den „normalen“ Abschnitten. Daher kann man Features, die aus Überschriften stammen, bspw. ein höheres Gewicht zuweisen als Features aus anderen „Regionen“ des Dokuments.[3]

1 Vgl. Weiss u.a. (2005), S. 25.
2 Vgl. Weiss u.a. (2005), S. 29.
3 Vgl. Weiss u.a. (2005), S. 31.

– Wichtigkeit der Features
  Eine andere Möglichkeit ist es, die Wichtigkeit eines Features für das betrachtete Dokument und die gesamte Dokumentsammlung zu berechnen, und diesen Wert als Gewicht des Features im Vektor des Dokuments einzutragen. Dafür wird zumeist das so genannte *term frequency - inverse document frequency (tf-idf)* Gewicht berechnet. Term frequency heißt so viel wie *Vorkommenshäufigkeit*. Um die Vorkommenshäufigkeit zu bestimmen, wird gezählt, wie oft ein Feature in einem Dokument vorkommt. Die inverse document frequency, also *inverse Dokumenthäufigkeit* gibt an, welche Bedeutung ein Feature für die Gesamtmenge der betrachteten Dokumente hat. Es wird ein Verhältnis zwischen der Vorkommenshäufigkeit eines Features in einem einzelnen Dokument und der invertierten Vorkommenshäufigkeit des Features über alle Dokumente gebildet. Für das Gewicht, das ein Feature erhält, bedeutet das also, dass es geringer wird, je häufiger ein Feature in der gesamten Dokumentsammlung auftaucht, und dass es im anderen Fall ansteigt.[1]

Hat man festgelegt, welche Art von Werten die Features erhalten sollen, so kann man für jedes Dokument der Dokumentsammlung einen Vektor erzeugen. Dafür wird die Menge aller Features betrachtet und für jedes Dokument überprüft, welche dieser Features in ihm enthalten sind. Diese werden dann aufgrund einer entsprechend vorher festgelegten Art mit einem Wert versehen und im Vektor des Dokuments gespeichert. Alle anderen Positionen des Vektors erhalten eine Null.
Da im Normalfall viele Positionen eines Vektors mit Nullen versehen werden[2], weil die entsprechenden Features nicht im zugehörigen Dokument vorhanden sind, erfolgt zumeist eine andere Speicherung der Vektoren. Die Nullen werden nicht gespeichert, sondern die eindeutige Identifikationsnummer des vorkommenden Features mit dem entsprechenden Wert.[3] Das bezeichnet man als „...*sparse vectors*...“[4]. Diese Repräsentation wird auch in der vorliegenden Arbeit verwendet.
Diese Vorgehensweise des Suchens der Features im Dokument und der entsprechenden Erzeugung des Vektors wird für alle Dokumente der Dokumentsammlung angewendet, so dass alle Dokumente über eine Repräsentation verfügen, die numerisch ist und weiterverarbeitet werden kann.

1 Für weitere Erläuterungen vgl. Weiss u.a. (2005), S. 30; Baeza-Yates u.a. (1999), S. 29; Manning u.a. (2008), S. 107-109.
2 Vgl. Feldman u.a. (2007), S. 4 f.
3 Bei binären Werten kann selbst dieser Wert weggelassen werden, da eine Angabe des Wertes überflüssig ist.
4 Weiss u.a. (2005), S. 31.

2.2 Ähnlichkeits- und Distanzmaße

2.2.1 Definition Ähnlichkeits- und Distanzmaß

Die Basis des Klassifizierens oder des Clusterns von Dokumenten ist die Ähnlichkeit zwischen ihnen. Dokumente sollen beim Klassifizieren Klassen zugewiesen werden, die Dokumente enthalten, die möglichst ähnlich zum neu zuzuweisenden Dokument sind. Beim Clustern sollen Dokumente gruppiert werden, die sich möglichst ähnlich sind und sich von den Dokumenten in anderen Clustern möglichst stark unterscheiden, zu diesen also unähnlich sind.[1] Um diese Ähnlichkeit zwischen Dokumenten bestimmen zu können, ist ein Ähnlichkeits- oder Distanzmaß nötig.
Ein „Ähnlichkeitskoeffizient“[2] stellt das Ergebnis einer mathematischen Funktion zur Ähnlichkeitsbestimmung zwischen zwei Objekten[3] als Zahl dar, die umso größer ist, je ähnlicher sich diese Objekte sind.[4]
Eine andere Möglichkeit, die Ähnlichkeit zwischen zwei Objekten zu messen, ist der umgekehrte Weg: Man misst die Unähnlichkeit oder Distanz zwischen den Objekten. Das Distanzmaß stellt die Unähnlichkeit zwischen zwei Objekten als Zahl dar, die umso größer ist, je unähnlicher sich die beiden Objekte sind.[5] Ein Ähnlichkeitskoeffizient, der auf den Bereich zwischen 0 und 1 normiert ist, kann deshalb immer in ein Distanzmaß umgewandelt werden, indem von der Zahl 1 die berechnete Ähnlichkeit subtrahiert wird.[6] Ein Distanzmaß erfüllt folgende Eigenschaften[7]:

- Identische Objekte haben eine Distanz von Null.
- Die Distanz zwischen zwei Objekten ist größer oder gleich Null, wobei die Reihenfolge der Objekte keine Rolle spielt.
- Die Dreiecksungleichung muss erfüllt sein, d.h., die direkte Distanz zwischen zwei Objekten ist nicht länger als die Distanz zwischen den Objekten, wenn ein drittes dazwischengeschaltet wird.[8]

1 Eine genauere Erläuterung des Klassifizierens befindet sich in Kapitel 2.3 und eine Erläuterung des Clusterns in Kapitel 2.4.
2 Handl (2010), S. 83.
3 Im Kontext dieser Arbeit handelt es sich bei den Objekten um Dokumente in der gewählten Dokumentrepräsentation.
4 Vgl. Handl (2010), S. 83.
5 Vgl. Handl (2010), S. 83.
6 Vgl. Handl (2010), S. 83.
7 Vgl. Eckey u.a. (2002), S. 205 f.
8 Laut Jain u.a. (1999), S. 273, muss dieser letzte Punkt nicht zwangsweise erfüllt sein. Im Folgenden werden aber nur Distanzmaße betrachtet, die diesen Punkt erfüllen.

2.2.2 Überblick über Ähnlichkeits- und Distanzmaße

2.2.2.1 Einführung

In der vorliegenden Arbeit werden die Dokumente, deren Ähnlichkeit oder Distanz berechnet werden soll, durch Feature-Vektoren dargestellt.[1] Diese Vektoren bilden die Grundlage für die Ähnlichkeits- und Distanzberechnungen, da bezogen auf sie als Repräsentanten der Dokumente alle Berechnungen ausgeführt werden. Betrachtet man also die Vektoren der Dokumente als Punkte in einem mehrdimensionalem Raum, so lässt sich die Ähnlichkeit oder die Distanz zwischen den Dokumenten als geometrischer Abstand zwischen ihren Vektoren definieren. Im Folgenden werden zwei Ähnlichkeits- und Distanzmaße vorgestellt, die in der vorliegenden Arbeit verwendet werden.[2]

2.2.2.2 Euklidisches Distanzmaß

Die euklidische Distanz zwischen zwei Feature-Vektoren $\vec{f}_{d_i} = (f_{1_{d_i}}, f_{2_{d_i}}, \ldots, f_{|F|_{d_i}})$ und $\vec{f}_{d_{i'}} = (f_{1_{d_{i'}}}, f_{2_{d_{i'}}}, \ldots, f_{|F|_{d_{i'}}})$ wird wie folgt bestimmt[3]:

$$\text{euclidean-distance}(\vec{f}_{d_i}, \vec{f}_{d_{i'}}) = \sqrt{\sum_{r=1}^{|F|} (f_{r_{d_i}} - f_{r_{d_{i'}}})^2} \tag{2.1}$$

1 Siehe Kapitel 2.1.

2 Neben den beiden in der vorliegenden Arbeit verwendeten Ähnlichkeits- und Distanzmaßen existieren weitere Maße, die zur Ähnlichkeitsbestimmung zwischen Dokumenten verwendet werden können. Beispiele finden sich in Handl (2010), S. 91 f., 94-97; Choi u.a. (2010), S. 44 f. Dabei handelt es sich um Ähnlichkeits- und Distanzmaße für binäre Feature-Vektoren, wie sie in der vorliegenden Arbeit verwendet werden, siehe Kapitel 2.1. Weitere Beispiele sind in Cha (2007), S. 301-304, 306 zu finden. Hierbei handelt es sich um Ähnlichkeits- und Distanzmaße für Wahrscheinlichkeitsdichtefunktionen, die teilweise auch in der Klassifizierung und im Clustern von Dokumenten eingesetzt werden können. Die beiden letzten Quellen: Forster (2006), S. 20-25; Strehl (2002), S. 92-95, erläutern insbesondere Distanzmaße, die beim Clustern von Dokumenten verwendet werden können.

3 Weitere Erläuterungen zum euklidischen Distanzmaß und zur Herleitung befinden sich bspw. in Handl (2010), S. 84-86. Die Formel selbst findet sich ebenfalls dort und auch in Mitchell (1997), S. 232.

2.2.2.3 Cosinusähnlichkeit

Die Cosinusähnlichkeit[1] basiert ebenfalls auf einer geometrischen Interpretation der Feature-Vektoren. Hierbei wird der Cosinus des Winkels zwischen den Vektoren bestimmt. Je kleiner dieser Winkel ist, desto größer ist die Ähnlichkeit zwischen den beiden Vektoren und desto kleiner die Distanz.[2]
Berechnet wird die Ähnlichkeit zwischen den beiden Feature-Vektoren $\vec{f}_{d_i} = (f_{1_{d_i}}, f_{2_{d_i}}, \ldots, f_{|F|_{d_i}})$ und $\vec{f}_{d_{i'}} = (f_{1_{d_{i'}}}, f_{2_{d_{i'}}}, \ldots, f_{|F|_{d_{i'}}})$ wie folgt[3]:

$$\begin{aligned} \text{cosine-similarity}(\vec{f}_{d_i}, \vec{f}_{d_{i'}}) &= \frac{\vec{f}_{d_i} \cdot \vec{f}_{d_{i'}}}{|\vec{f}_{d_i}||\vec{f}_{d_{i'}}|} \\ &= \frac{\sum_{r=1}^{|F|} f_{r_{d_i}} * f_{r_{d_{i'}}}}{\sqrt{\sum_{r=1}^{|F|} f_{r_{d_i}}^2} * \sqrt{\sum_{r=1}^{|F|} f_{r_{d_{i'}}}^2}} \end{aligned} \tag{2.2}$$

2.3 Klassifizieren

2.3.1 Darstellung des Klassifizierens

Unter dem Vorgang des Klassifizierens[4] versteht man die Zuordnung von Objekten zu einer endlichen Menge vordefinierter Klassen.[5]
In dieser Arbeit wird nicht die manuelle Form der Klassifizierung betrachtet, sondern die automatisierte, von Computern durchgeführte Form. Insbesondere steht der Ansatz des *machine learning* (deutsch: maschinellen Lernens) im Vordergrund, bei dem das Kriterium zur Entscheidung über die Klassenzuordnung aus Trainingsdaten erlernt wird, die zuvor jedoch manuell klassifiziert worden sind.[6]

1 Im Gegensatz zum euklidischen Distanzmaß handelt es sich bei der Cosinusähnlichkeit nicht um ein Distanzmaß, sondern um ein Ähnlichkeitsmaß. Eine Überführung in ein Distanzmaß ist jedoch möglich, indem der berechnete Wert von eins abgezogen wird, wenn die Cosinusähnlichkeit auf den Bereich zwischen 0 und 1 normiert wurde. Das ist in der vorliegenden Arbeit der Fall, wenn die Ähnlichkeit in eine Distanz umgewandelt wird.
2 Vgl. Manning u.a. (2008), S. 112.
3 Vgl. Manning u.a. (2008), S. 111. Der Zähler ist das Skalarprodukt der beiden Vektoren und der Nenner das Produkt der euklidischen Normen beider Vektoren, also ihrer Längen, vgl. Manning u.a. (2008), S. 111.
4 Die Begriffe Kategorisierung oder Labelling, die zum Teil synonym zu Klassifizierung verwendet werden, werden hier nicht benutzt. Stattdessen werden die Begriffe Klassifizieren oder Klassifizierung verwendet.
5 Vgl. Manning u.a. (2008), S. 234
6 Vgl. Manning u.a. (2008), S. 236

Es werden natürlichsprachliche Texte, also Dokumente, betrachtet. Diese sind die oben erwähnten Objekte, die, basierend auf ihrem Inhalt[1], klassifiziert werden.[2] Die Menge der Dokumente sei $D = \left\{d_1, d_2, ..., d_{|D|}\right\}$, wobei $|D|$ die Anzahl der Dokumente der Menge D meint. Die Menge der Klassen sei $L = \left\{L_1, L_2, ..., L_{|L|}\right\}$, wobei $|L|$ die Anzahl der Klassen in L meint. Nun wird eine Funktion gesucht, die jeder Klasse die entsprechenden Dokumente zuordnet. Genauer gesagt wird, nach Sebastiani (2002)[3], eine Approximation der Zielfunktion $\breve{\Phi}$ gesucht. Die Approximation ist die Funktion Φ, auch Klassifizierer genannt, die jeder Kombination aus Dokument und Klasse $\langle d_i, L_j \rangle \in D \times L$[4] entweder den Wert *true* oder den Wert *false* zuordnet. Ersteres bedeutet, dass das Dokument zur Klasse gehört, letzteres das Gegenteil. D.h., es wird ein Klassifizierer

$$\Phi : D \times L \rightarrow \{true, false\}$$

gesucht, der die unbekannte Zielfunktion

$$\breve{\Phi} : D \times L \rightarrow \{true, false\}$$

sehr gut[5] approximiert. Dafür ist eine Menge TR von Trainingsdokumenten gegeben, die bereits manuell klassifiziert wurden. Mit Hilfe dieser Klassifikation der Trainingsdokumente und eines Klassifizierungsalogrithmus wird eine Klassifizierungsfunktion erlernt, die die Dokumente einer Menge TE von Testdokumenten klassifiziert. Der Ablauf einer Klassifizierung wird in Abbildung 2.2 dargestellt.[6]

1 Mit Inhalt eines Dokuments ist nicht der für den Menschen verstehbare Inhalt, also die Semantik des Textes, gemeint, sondern der für einen Computer erfassbare Inhalt, also die enthaltenen Zeichensequenzen. Klassifizierungen, die auf dem Inhalt der Dokumente basieren, berücksichtigen also keine Metainformationen, sondern nur die im Dokument enthaltenen Zeichensequenzen.

2 Vgl. Sebastiani (2002), S. 1; Yang (1997), S. 69.

3 Vgl. Sebastiani (2002), 2 f.

4 Das kartesische Produkt $A \times B$ der beiden Mengen A und B besteht aus der Menge aller geordneten Paare aus A und B, also $A \times B = \{(a, b) | a \in A, b \in B\}$, vgl. Deiser (2010), S. 50.

5 Zur Definition und Messung dieses „sehr gut“ siehe Kapitel 2.5 dieser Arbeit.

6 Dabei ist der grobe Ablauf Kehagias u.a. (2003), S. 228, entnommen und durch einige Ergänzungen der Verfasserin verfeinert. Die Vorverarbeitung wird in Kapitel 2.1.2 erläutert.

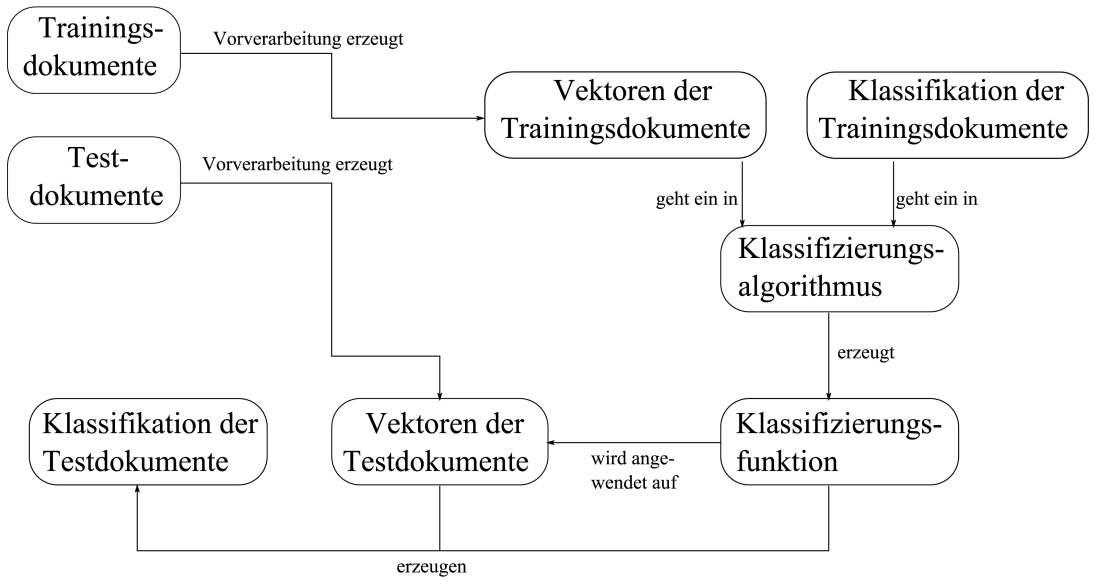


Abbildung 2.2: Ablauf einer Klassifizierung

Beim Klassifizieren geht man von der Annahme aus, dass beide Mengen aus der gleichen Dokumentenmenge D stammen.[1] Es gilt also:

$$TR \subset D$$
$$TE \subset D$$
$$TR \cap TE = \emptyset$$

Ziel des Klassifizierens ist es, die Menge der Testdokumente mit einer hohen Genauigkeit[2] den richtigen Klassen zuzuordnen, damit die Klassifizierungsfunktion auch auf neue Dokumente angewendet werden kann. Diese Art des Lernens bezeichnet man als *supervised* (deutsch: überwacht), da hier ein Mensch die Trainingsdaten klassifiziert und somit in den Lernprozess eingreift.[3]

1 Vgl. Manning u.a. (2008), S. 238. Das beinhaltet eine große Ähnlichkeit zwischen Dokumenten der Trainings- und der Testmenge, die über die vorgestellten Ähnlichkeits- und Distanzmaße messbar ist. Wäre das nicht der Fall, so könnte kein Klassifizierer erlernt werden, der unbekannte Testdokumente klassifizieren kann. Es handelt sich jedoch nicht um dieselben Dokumente.

2 Siehe hierzu Kapitel 2.5.

3 Zur Abgrenzung zwischen überwachtem und unüberwachtem Lernen vgl. bspw. Segaran (2007), S. 29 f.

Zusätzlich wird Folgendes vorausgesetzt[1]:

- Bei den Klassennamen handelt es sich um eine Art „Beschriftung“. Sie enthalten also keine zusätzlichen Informationen, die während des Klassifizierens verwendet werden. Insbesondere wird der Text der „Beschriftung“ nicht verwendet.
- Es werden nur *endogene* Informationen der Dokumente während des Klassifizierens verwendet. Zusätzliche Metadaten werden also nicht berücksichtigt, das Klassifizieren beruht allein auf dem Inhalt des Dokuments.

2.3.2 Klassifizierungsansätze

2.3.2.1 Zuordnung der Dokumente zu genau einer Klasse

2.3.2.1.1 Definition der Zuordnung zu genau einer Klasse

Der Fall der Zuordnung jedes Dokuments zu genau einer Klasse wird „*single-label*“[2] oder auch *nichtüberlappend* (englisch: *nonoverlapping*) genannt. Der Klassifizierer Φ weist hier genau einer Kombination aus Klasse und Dokument den Wert *true* zu, alle anderen Kombinationen aus diesem Dokument mit anderen Klassen aus der Menge der möglichen Klassen erhalten den Wert $false$.

2.3.2.1.2 Zuordnung zu genau einer Klasse ohne Hierarchie

Die Zuordnung der Dokumente innerhalb des Klassifizierens erfolgt anhand der gelernten Entscheidungen des auf den Trainingsdaten trainierten Klassifizierers. Ohne Hierarchie bedeutet in diesem Fall, dass die den Trainingsdaten zu Grunde liegende Klassifikation, anhand derer sie manuell klassifiziert wurden, keine hierarchische Struktur besaß. Also gilt die Annahme, dass auch die zu klassifizierenden Dokumente nicht in eine hierarchische Struktur eingeordnet werden müssen und demzufolge die für das Klassifizieren verwendeten Algorithmen nicht auf die Beachtung einer solchen Struktur ausgerichtet sein müssen.
Die Dokumente sollen genau einer Klasse der vorhandenen Klassifikation zugeordnet werden. Hierbei können zwei Fälle unterschieden werden.

1. Die Entscheidung des Klassifizierers muss zwischen einer Klasse und ihrer Komplementklasse getroffen werden.

1 Vgl. Sebastiani (2002), S. 3.
2 Sebastiani (2002), S. 3. Eine weitere Bezeichnung ist *one-of*, vgl. Manning u.a. (2008), S. 238. In der vorliegenden Arbeit wird *single-label* als Bezeichnung für die Zuordnung von Dokumenten zu genau einer Klasse verwendet.

2. Die Entscheidung des Klassifizierers muss zwischen mindestens zwei Klassen getroffen werden.

Im ersten Fall spricht man von einer *binären* Klassifizierung. Hier existiert nur eine Klasse und jedes Dokument wird ihr zugeordnet oder nicht.[1] Bei der Zuordnung handelt es sich um die Entscheidungen „Objekt gehört zur Klasse“ und „Objekt gehört nicht zur Klasse“. Das bedeutet, jedem Dokument wird, bezogen auf die Klasse, entweder ein *true* oder ein *false* zugeordnet.

Wird dagegen nicht ein Klassifizierer erzeugt, der eine „ja“ oder „nein“ Entscheidung in Bezug auf eine Klasse für alle zu klassifizierenden Dokumente trifft, sondern der Klassifizierer entscheidet, zu welcher der vorhandenen Klassen die Dokumente gehören, so spricht man nicht mehr von einer binären Klassifizierung, sondern von einer *multi-class* Klassifizierung. Auch hier wird jedem Dokument nur *eine* der möglichen Klassen zugeordnet, jedoch wird diese aus der Menge der möglichen Klassen ausgewählt. Diese Auswahl selbst kann direkt durch den Klassifizierer erfolgen, indem bspw. zunächst ein Ranking[2] der möglichen Klassen für jedes Dokument durch den Klassifizierer erzeugt wird und dann jeweils die Klasse, die an erster Stelle des Rankings steht, zugeordnet wird. Jedes single-label multi-class Klassifizierungsproblem kann wieder auf eine single-label binäre Klassifizierung zurückgeführt werden. Dafür schlagen Manning u.a. (2008) vor, dass ein binärer Klassifizierer für jede der möglichen Klassen erzeugt wird und die Trainingsbeispiele entsprechend pro Klasse ein Label erhalten, ob sie zu dieser gehören oder nicht.[3] Alle erzeugten Klassifizierer werden dann nacheinander auf die Testdokumente angewendet und ermitteln, ob das Testdokument zur Klasse gehört oder nicht. Abschließend wird ein Wert[4] für jede der zugeordneten Klassen berechnet und schließlich diejenige Klasse zugeordnet, deren Wert der höchste ist.

Zumeist wird bei der Beschreibung von Klassifizierungsalgorithmen von einer Zuordnung zu einer Klasse oder vom binären Fall ausgegangen, weil es sich hier um den allgemeineren Fall handelt, da die Transformationsmöglichkeit in ein binäres Problem immer gegeben ist.[5]

1 Vgl. Sebastiani (2002), S. 3.
2 Unter einem Ranking versteht man das Aufstellen einer Reihenfolge der möglichen Klassen, wobei die Sortierung der Klassen absteigend erfolgt. Das bedeutet, die Klasse an der Spitze des Rankings ist die am besten passende Klasse.
3 Vgl. Manning u.a. (2008), S. 282 f.
4 Dabei kann es sich bspw. um einen Konfidenzwert oder eine Wahrscheinlichkeit handeln, vgl. Manning u.a. (2008), S. 283.
5 Vgl. Sebastiani (2002), S. 3 f.

2.3.2.1.3 Zuordnung zu genau einer Klasse mit Hierarchie

Klassifikationen sind oftmals hierarchisch aufgebaut. So können zunächst generelle Unterscheidungen getroffen werden, die von Subklassen weiter ausdifferenziert werden. Auch bei der Klassifizierung von Dokumenten können die vorhandenen Klassen hierarchisch angeordnet sein. Laut Manning u.a. (2008) kann eine solche Hierarchie innerhalb des Klassifizierens genutzt werden, jedoch meinen die Autoren, dass Ansätze in dieser Richtung bisher keine wirklich überzeugenden Ergebnisse geliefert haben.[1]
Problematisch ist bspw. dass oftmals nicht genau definiert wird, welche Evaluationsmaße verwendet werden und wie Experimente in Bezug auf hierarchische Klassifizierung durchzuführen sind.[2] Silla u.a. (2011) definieren zunächst, was eine hierarchische Klassifikation ist[3]:

- Die vordefinierte Klassifikation ist entweder als Baum oder als Graph aufgebaut.
  Der Unterschied besteht darin, dass die Klassen in einem Baum maximal eine Vaterklasse haben und in einem Graph mehr als eine Vaterklasse haben können. Durch das Vorhandensein mehr als einer Vaterklasse wird die Komplexität der Hierarchie erhöht.

- Die Klassen sind über eine „IS-A“-Beziehung miteinander verbunden.
  Das bedeutet, die Klassen sind Spezialisierungen voneinander, wenn man im Baum oder im Graph in die nächste Ebene absteigt, und Generalisierungen voneinander, wenn man im Baum oder im Graph in die übergeordnete Ebene aufsteigt.

Hierarchische Klassifizierungsprobleme sind demnach inhärent multi-class und multi-label[4] Probleme, da, sobald ein Dokument einer Klasse der Hierarchie zugeordnet wird, auch alle Vaterklassen zugeordnet werden müssen, weil diese die Generalisierungen der zugeordneten Klasse sind und durch die „IS-A“-Beziehung die Dokumente auch zu den übergeordneten Vaterklassen gehören. In der vorliegenden Arbeit wird jedoch wie in Silla u.a. (2011)[5] eine hierarchische Klassifizierung nur dann als multi-label bezeichnet, wenn nicht nur die Vaterklassen zugeordnet werden, sondern der Klassifizierer auch die Möglichkeit hat, außerhalb des Hierarchiezweigs weitere Klassen zuzuordnen.

1 Vgl. Manning u.a. (2008), S. 310.
2 Vgl. Silla u.a. (2011), S. 32.
3 Vgl. Silla u.a. (2011), S. 34.
4 Siehe Kapitel 2.3.2.2.
5 Vgl. Silla u.a. (2011), S. 35.

Es existieren verschiedene Ansätze, um Dokumente einer Klasse einer Hierarchie zuzuordnen[1]:

1. Erstellen eines lokalen Klassifizierers für jeden Knoten der Hierarchie
   Hierbei werden binäre Klassifizierer pro Klasse der Hierarchie anhand der Trainingsdaten erzeugt und auf die Testdaten angewendet. Nachteilig ist, dass es in diesem Fall zu Inkonsistenzen kommen kann, wenn bspw. eine Kindklasse zugeordnet wird, die dazugehörige Vaterklasse jedoch nicht.

2. Erstellen eines lokalen Klassifizierers für jeden inneren Knoten der Hierarchie
   Das bedeutet, für die Blattknoten wird kein eigener Klassifizierer erzeugt. Die Klassifizierer können in diesem Fall binäre oder multi-class Klassifizierer sein. Sobald ein Vaterknoten von einem Klassifizierer als Klasse festgelegt wurde, werden nur noch dessen Kindknoten betrachtet und keine anderen Zweige der Hierarchie mehr. Hierbei können die zuvor genannten Inkonsistenzen nicht auftreten.

3. Erstellen eines lokalen Klassifizierers für jede Hierarchieebene
   Pro Hierarchieebene wird ein binärer oder multi-class Klassifizierer erzeugt und auf die Testdokumente angewendet. Dabei sind wieder die zuvor genannten Inkonsistenzen möglich.

4. Erstellen eines globalen Klassifizierers
   Dieser Klassifizierer erzeugt aus den Trainingsdaten ein Modell, das die gesamte Hierarchie beachtet. Den Testdokumenten können somit Klassen jeder Ebene zugeordnet werden und Inkonsistenzen werden vermieden. Bisher ist dieser Ansatz laut Silla u.a. (2011) noch nicht klar definiert worden.[2]

2.3.2.2 Zuordnung der Dokumente zu mehreren Klassen

2.3.2.2.1 Definition der Zuordnung zu mehreren Klassen

Kann ein Dokument mehreren Klassen zugeordnet werden, so bezeichnet man das als „*multilabel*“[3] oder *überlappendes* (englisch: *overlapping*) Klassifizieren. Es wird also eine Teilmenge der vorhandenen endlichen Menge an Klassen zugeordnet. Es

1 Vgl. Silla u.a. (2011), S. 37-50.
2 Vgl. Silla u.a. (2011), S. 48.
3 Sebastiani (2002), S. 3. Eine weitere Bezeichnung ist *any-of*, vgl. Manning u.a. (2008), S. 238. In der vorliegenden Arbeit wird die Bezeichnung *multi-label* verwendet, wenn von der Zuordnung von Dokumenten zu mehr als einer Klasse die Rede ist. Die von Sebastiani (2002) abweichende Schreibweise wird gewählt, damit die Schreibweise der im Fall der Zuordnung von Dokumenten zu genau einer Klasse entspricht.

besteht daher die Möglichkeit, dass ein Dokument nur einer Klasse oder auch keiner der vordefinierten Klassen zugeordnet wird.[1] Das bedeutet, eine Zuordnung eines Dokuments zu einer Klasse führt nicht dazu, dass dem Dokument keine weitere Klasse aus der Menge der Klassen zugeordnet werden kann.
Zudem ist es hier möglich, einen Klassifizierer zu erzeugen, der nicht nur angibt, ob ein Dokument zu einer Klasse gehört, sondern auch zu welchem Grad. D.h., der Klassifizierer kann bspw. für ein Dokument einen Prozentsatz für die Zugehörigkeit zu einer Klasse angeben.
Eine multi-label Klassifizierung ist inhärent eine multi-class Klassifizierung. Ohne mehrere Klassen zuordnen zu können, wie es bspw. bei der binären Klassifizierung der Fall ist, würde man auch nicht von einer multi-label Klassifizierung sprechen.

2.3.2.2.2 Zuordnung zu mehreren Klassen ohne Hierarchie

Sind die Klassen, denen die Dokumente zugeordnet werden sollen, nicht hierarchisch aufgebaut und kann ein Dokument mehreren Klassen zugeordnet werden, so handelt es sich um eine Zuordnung zu mehreren Klassen.
Obwohl in diesem Fall eine Zuordnung eines Dokuments zu mehreren Klassen möglich ist, spielt auch die binäre Klassifizierung eine Rolle im Bereich der multi-label Klassifizierung. So ist eine beschriebene Vorgehensweise, um eine multi-label Klassifizierung durchzuführen, die folgende: Man erstellt wieder einen binären Klassifizierer für jede der möglichen Klassen, der anhand der Trainingsdokumente gelernt hat zu entscheiden, ob ein Dokument zu dieser Klasse gehört oder nicht, und wendet jeden der erzeugten Klassifizierer auf alle Testdokumente an.[2] Für jedes Testdokument legt jeder Klassifizierer fest, ob das Dokument zur betrachteten Klasse gehört oder nicht. Alle Klassen, denen das Dokument über alle Klassifizierer zugeordnet wurde, bilden dann die Menge der Klassen des Dokuments. Es wird also keine weitere Auswahl unter den Klassen getroffen und eine Entscheidung eines einzelnen Klassifizierers für oder gegen eine Klassenzugehörigkeit eines Testdokuments beeinflusst die Entscheidungen der anderen Klassifizierer nicht. Es findet also eine Transformation des multi-label Problems in unabhängige binäre Probleme statt, die dann wiederum mit Algorithmen für den binären Fall gelöst werden können.[3]
Die Transformation lässt sich auch erweitern, indem das multi-label Problem nicht in binäre Probleme zerlegt wird, sondern generell in ein oder mehrere single-label

1 Vgl. Manning u.a. (2008), S. 281.
2 Vgl. bspw. Manning u.a. (2008), S. 282; Sebastiani (2002), S. 3 f.
3 Vgl. bspw. Read u.a. (2008), S. 995; Read u.a. (2009), S. 255; Sajnani u.a. (2011), S. 58; Zhang u.a. (2005), S. 718.

Probleme. In diesem Fall kann die gleiche Vorgehensweise beim Klassifizieren gewählt werden, wie für den single-label Fall beschrieben.[1] Dazu gehört die binäre Klassifizierung als eine mögliche Transformation. Als weitere Möglichkeit existiert das bei der single-label Klassifizierung genannte Ranking[2], aber auch eine spezielle Transformation für den multi-label Fall, die darauf basiert, aus jeder Teilmenge an Klassen, die in den Trainingsdokumenten auftaucht, eine künstliche Klasse zu erzeugen und für diese künstlichen Klassen einen Klassifizierer zu trainieren. Dieser ordnet den Testdokumenten die künstlichen Klassen zu, die dann wieder in die Teilmengen der Originalklassen zerlegt werden können.[3]

Alle drei Transformationsansätze haben Vor- und Nachteile. Die Überführung eines multi-label Problems in ein binäres oder ein Ranking-Problem hat den Vorteil, dass jede Kombination aus Klassen den Testdokumenten zugeordnet werden kann. Der Nachteil daran ist, dass angenommen wird, die Klassen seien voneinander unabhängig, d.h., die Zuordnung einer bestimmten Klasse beeinflusst nicht die Zuordnung von anderen Klassen. Es kann aber durchaus in der Realität der Fall sein, dass die Zuordnung einer Klasse die Zuordnung anderer Klassen beeinflusst.[4] Künstliche Klassen aus Teilmengen von Klassen zu erzeugen beachtet Abhängigkeiten zwischen den Klassenzuordnungen. Nachteilig an diesem Ansatz ist jedoch, dass in den Trainingsdaten durchaus ungewöhnliche Kombinationen aus Klassen vorhanden sein können, die nicht generell in Testdokumenten auftauchen, und dass auch nur solche Kombinationen zugeordnet werden können, die von den Trainingsdokumenten abgebildet werden.[5]

Eine weitere Möglichkeit, das multi-label Problem zu lösen, ist die Anpassung eines single-label Algorithmus, so dass der entstehende Klassifizierer nicht nur genau eine Klasse zuordnen kann.[6]

Daneben wird versucht, durch speziell auf multi-label Probleme zugeschnittene Vorgehensweisen nicht mehr das gesamte Problem zu transformieren, sondern neue Algorithmen für das multi-label Problem zu entwickeln. Da die Experimente in der vorliegenden Arbeit nur die single-label Klassifizierung betrachten, diese Ansätze

1 Vgl. bspw. Read u.a. (2008), S. 995; Read u.a. (2009), S. 254.
2 Vgl. bspw. Read u.a. (2008), S. 995.
3 Vgl. bspw. Read u.a. (2008), S. 995; Sajnani u.a. (2011), S. 59.
4 Es ist nicht abschließend geklärt, ob die Transformation in binäre Teilprobleme wirklich darunter leidet, dass diese eventuellen Beziehungen zwischen den Klassen nicht berücksichtigt werden. Es gibt in der Literatur auch Autoren, die das nicht als gravierenden Nachteil sehen, vgl. bspw. Read u.a. (2009), 255 f.
5 Vgl. bspw. Read u.a. (2008), 995 f.
6 Vgl. bspw. Read u.a. (2008), S. 995; Read u.a. (2009), S. 254; Sajnani u.a. (2011), S. 59; Zhang u.a. (2005), S. 718.

sehr neu sind[1] und bisher nicht nachhaltig evaluiert wurden, wird an dieser Stelle nicht ausführlicher auf multi-label Probleme eingegangen, sondern auf die entsprechende Literatur verwiesen.[2]

2.3.2.2.3 Zuordnung zu mehreren Klassen mit Hierarchie

Die Definition der Hierarchie bleibt die gleiche, wie in Kapitel 2.3.2.1.3 gegeben. Der Unterschied besteht nun darin, dass nicht nur die Vaterklassen der zugeordneten Klasse ebenfalls zugeordnet werden, sondern dass auch Klassen anderer Zweige der Hierarchie zugeordnet werden können.
Die oben gegebenen Ansätze für ein solches Klassifizieren müssen also wie folgt angepasst werden[3]:

1. Erstellen eines lokalen Klassifizierers für jeden Knoten der Hierarchie
   Es kann weiterhin ein binärer Klassifizierer für jeden Knoten der Hierarchie erzeugt werden. Dieser Ansatz ist direkt auf das multi-label Problem anwendbar, da alle Klassen, die von den Klassifizierern vorgeschlagen werden, auch zugeordnet werden können und nicht mehr nur eine Klasse - und deren Vaterklassen - ausgewählt werden muss.

2. Erstellen eines lokalen Klassifizierers für jeden inneren Knoten der Hierarchie
   Dieser Ansatz ist nicht direkt auf das multi-label Problem anwendbar, da in dem Ansatz andere Zweige der Hierarchie direkt ausgeschlossen werden. Hier müsste entweder ein mutli-label Klassifizierer für jeden Elternknoten erzeugt werden oder aber mit Schwellenwerten gearbeitet werden.[4]

3. Erstellen eines lokalen Klassifizierers für jede Hierarchieebene
   Der Ansatz kann direkt mit einem multi-label Klassifizierer genutzt werden.

4. Erstellen eines globalen Klassifizierers
   Auch dieser Ansatz kann direkt genutzt werden.

In der Praxis ist es in den meisten Fällen so, dass multi-label Probleme in single-label Probleme transformiert werden, um die Komplexität geringer zu halten.[5] Daneben

1 Vgl. Sajnani u.a. (2011), S. 57.
2 Als Beispiele sind zu nennen: Read u.a. (2008); Read u.a. (2009); Zhang u.a. (2005); Younes (2011); Dembczynski u.a. (2010); Alfaro u.a. (2011); Rivera u.a. (2011); Ghamrawi u.a. (2005); Ji u.a. (2008); Tsoumakas u.a. (2007b).
3 Vgl. Silla u.a. (2011), S. 37-50.
4 Vgl. Silla u.a. (2011), S. 44.
5 Vgl. Silla u.a. (2011), S. 52.

existieren aber auch Ansätze, Algorithmen abzuwandeln, damit diese die hierarchischen multi-label Probleme direkt lösen können. Beispiele hierfür sind: Vens u.a. (2008) und Esuli u.a. (2008).

2.3.3 Klassifizierungsalgorithmen

2.3.3.1 Naive-Bayes-Algorithmus

2.3.3.1.1 Der Satz von Bayes

Der Naive-Bayes-Algorithmus basiert auf dem Satz von Bayes[1]. Dieser besagt, wie man mit bedingten Wahrscheinlichkeiten rechnet. Das heißt[2],

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

bedeutet: Die Wahrscheinlichkeit für das Ereignis A ist, unter der Bedingung, dass Ereignis B eingetreten ist, die Wahrscheinlichkeit für B, wenn A eingetreten ist, multipliziert mit der A-priori-Wahrscheinlichkeit von A, dividiert durch die A-priori-Wahrscheinlichkeit von B.

„A-priori-Wahrscheinlichkeit“ bezeichnet dabei einen Wahrscheinlichkeitswert, der aufgrund von Vorwissen, jedoch ohne weitere Informationen über das Ereignis zu haben, gewonnen wurde. Das bedeutet, diese Wahrscheinlichkeit gibt den Grad an, mit dem man an das Eintreten des betreffenden Ereignisses glaubt, ohne irgendeine weitere Information zu haben.[3]

Einsatzgebiete für den Satz von Bayes sind Umkehrschlussfolgerungen. Das bedeutet, man besitzt häufig die Wahrscheinlichkeit für P(Ereignis|Ursache), benötigt aber die Wahrscheinlichkeit für P(Ursache|Ereignis). Es wird die so genannte „A-posteriori“-Wahrscheinlichkeit der Ursache berechnet, wenn man weiß, dass ein Ereignis eingetreten ist. Dabei handelt es sich um eine Relativierung des Glaubens an das Eintreten der Ursache unter der Voraussetzung, dass man nun weiß, dass das Ereignis eingetreten ist, und sonst keine weiteren Informationen bezüglich der Ursache besitzt.[4]

1 Vgl. Bayes u.a. (1763).
2 Die Prämissen sind: $P(A) \neq 0$ und $P(B) \neq 0$.
3 Vgl. Russell u.a. (2003), S. 468.
4 Vgl. Russell u.a. (2003), S. 470.

2.3.3.1.2 Übertragung des Satzes von Bayes auf die Klassifizierung von Texten

Übertragen auf die Klassifizierung von Texten ist die Wahrscheinlichkeit für eine Klasse bezogen auf ein bestimmtes Dokument gesucht: $P(L_j|d_i)$. Würde man den Satz von Bayes direkt anwenden, so ergäbe sich:

$$P(L_j|d_i) = \frac{P(d_i|L_j) * P(L_j)}{P(d_i)}.$$

Das wiederum setzt voraus, dass die A-priori-Wahrscheinlichkeiten für die einzelnen Klassen, für das Dokument sowie für das Dokument, wenn eine bestimmte Klasse betrachtet wird, bekannt sein müssen. Man legt hier eine Maximum-likelihood-Schätzung zu Grunde, indem man die relative Häufigkeit als A-priori-Wahrscheinlichkeit verwendet.[1] Das bedeutet für eine Klasse:

$$P(L_j) = \frac{|L_j|}{|D|}$$

wobei $|L_j|$ die Anzahl der Dokumente der Klasse L_j meint und $|D|$ die Gesamtanzahl der Dokumente.[2]
Die Dokumente werden hier durch Vektoren repräsentiert[3], die die An- oder Abwesenheit von Features in diesen Dokumenten darstellen. Aus diesem Grund erfolgt die Berechnung der A-priori-Wahrscheinlichkeiten für die Dokumente über die Vektoren

1 Vgl. Manning u.a. (2008), S. 240. Hier wird eine Wahrscheinlichkeit vorhergesagt. In Mitchell (1997), S. 167-170, wird gezeigt, dass Maximum-likelihood-Schätzungen für die Vorhersage von Wahrscheinlichkeiten verwendet werden können. Laut Russell u.a. (2003), S. 715, ist es insbesondere ein geeigneter Ansatz, wenn a priori keine der Klassen wahrscheinlicher ist als eine andere.

2 Vgl. Manning u.a. (2008), S. 240; Weiss u.a. (2005), S. 67. Genauer wäre eigentlich von der Gesamtanzahl der Trainingsdokumente $|TR|$ zu sprechen. In der Literatur, vgl. Manning u.a. (2008), S. 240; Weiss u.a. (2005), S. 67, wird jedoch von der Gesamtanzahl der Dokumente gesprochen, da bei einer Verwendung des Naive-Bayes-Algorithmus in einem Szenario, bei dem es nicht um Experimente, sondern um den wirklichen Einsatz geht, keine Einteilung in Trainings- und Testdaten erfolgt, sondern alle vorhandenen Dokumente als Trainingsdokumente verwendet werden.

3 Siehe Kapitel 2.1 dieser Arbeit.

dieser oder, genauer gesagt, über die Werte der Komponenten des Vektors[1] dieser Dokumente.[2] Man berechnet also die A-priori-Wahrscheinlichkeit dafür, dass ein Dokument d_i in der Klasse L_j vorkommt, indem die Wahrscheinlichkeit dafür berechnet wird, dass der Vektor $\vec{f}_{d_i}$ als Repräsentant des Dokuments d_i in Klasse L_j vorkommt.

$$P(d_i|L_j) = P(\vec{f}_{d_i}|L_j)$$

Die Berechnung der Wahrscheinlichkeit, dass der Vektor $\vec{f}_{d_i}$ in der Klasse L_j vorkommt, kann aufgrund der Unabhängigkeitsannahme der Features und damit der Komponenten des Vektors $\vec{f}_{d_i}$ folgendermaßen dargestellt werden:

$$P(\vec{f}_{d_i}|L_j) = \prod_{r=1}^{|F|} \left[P(f_r = 0|L_j) * \left(\frac{P(f_r = 1|L_j)}{P(f_r = 0|L_j)} \right)^{f_{r_{d_i}}} \right] \tag{2.3}$$

Die Wahrscheinlichkeit $P(f_r = 0|L_j)$ ist dabei die Wahrscheinlichkeit, dass ein Feature f_r in der Klasse L_j, also in mindestens einem Trainingsdokument d_a, welches zur Klasse L_j gehört, nicht enthalten ist, und $P(f_r = 1|L_j)$ meint den umgekehrten Fall, nämlich dass das Feature f_r enthalten ist. Die Wahrscheinlichkeiten werden berechnet, indem man die Anzahl der Vektoren bestimmt, für die jeweils diese Werte gelten, und sie durch die Anzahl der in der betrachteten Klasse vorhandenen Vektoren dividiert.

1 Die in diesem Kapitel beschriebene Herleitung bezieht sich auf binäre Feature-Vektoren, die nicht *sparse* sind.

2 Hier wird die Annahme getroffen, dass die Features unabhängig voneinander sind, vgl. Weiss u.a. (2005), S. 67, und auch positionsunabhängig sind, vgl. Manning u.a. (2008), S. 247. Daher kann die bedingte Wahrscheinlichkeit über das Produkt der einzelnen Wahrscheinlichkeiten für das Vorkommen eines Features in einer Klasse berechnet werden, vgl. Mitchell (1997), S. 165, 177. Die Annahme der Unabhängigkeit ist in den seltensten Fällen begründet. Im Gegenteil, gerade bei natürlichsprachlichen Texten spielen sowohl die Reihenfolge der Worte, wenn man diese als Features betrachtet, als auch die Abhängigkeit der Features voneinander in Bezug auf die Klassenzugehörigkeit eine große Rolle. So werden einige Featurekombinationen in bestimmten Klassen häufiger vorkommen als in anderen Klassen, vgl. Manning u.a. (2008), S. 248 f. Vereinfachend wird davon ausgegangen, dass die Features unabhängig voneinander sind und die Methode wird deshalb als „naiv“ bezeichnet. Trotzdem lässt sich ein Naive-Bayes-Klassifizierer gut verwenden. Er schätzt zwar aufgrund der teilweise falschen Unabhängigkeitsannahmen die tatsächlichen Wahrscheinlichkeiten nicht sehr gut, jedoch klassifiziert er trotzdem gut, vgl. Manning u.a. (2008), S. 249; Russell u.a. (2003), S. 482. Hierfür ist nur entscheidend, dass die korrekte Klasse die höchste Wahrscheinlichkeit erhält. Das ist bei dieser Methode größtenteils der Fall.

Ausgedrückt wird das in den Formeln 2.4 und 2.5, deren Zähler jeweils die Anzahl der Vektoren $\vec{f}_{d_a}$ bestimmt, für die gilt: a ist ein Laufindex, so dass jeder Vektor, der ein Dokument d_a aus D repräsentiert, betrachtet wird, und jedes Dokument d_a gehört zur Klasse L_j und der Wert des Features $f_{r_{d_a}}$ ist entweder 0 oder 1.

$$P(f_r = 0|L_j) = \frac{\left|\left\{\vec{f}_{d_a} \middle| a = 1..|D| \wedge d_a \in L_j \wedge f_{r_{d_a}} = 0\right\}\right|}{|L_j|} \text{ und} \tag{2.4}$$

$$P(f_r = 1|L_j) = \frac{\left|\left\{\vec{f}_{d_a} \middle| a = 1..|D| \wedge d_a \in L_j \wedge f_{r_{d_a}} = 1\right\}\right|}{|L_j|} \tag{2.5}$$

Zusätzlich taucht in der Formel 2.3 der Exponent $f_{r_{d_i}}$ auf. Dieser bezieht sich auf den Wert des Features f_r im betrachteten Dokument d_i. Für dieses soll die Wahrscheinlichkeit berechnet werden, dass es zur Klasse L_j gehört. Dafür müssen die Werte seiner Features herangezogen werden. Das geschieht, indem der Bruch aus Formel 2.3 mit dem Featurewert des Dokuments potenziert wird. Taucht das Feature f_r als 0 im Dokument d_i auf, so wird aus dem Bruch insgesamt eine 1 und nur die Wahrscheinlichkeit dafür, dass das Feature nicht in der Klasse enthalten ist, geht in das Produkt über alle Features ein. Im umgekehrten Fall bleibt der Bruch bestehen und die beiden Wahrscheinlichkeiten für das Nicht-Vorhandensein des Features im Produkt heben sich auf.
Insgesamt ergibt sich also für die Wahrscheinlichkeit einer Klasse L_j für ein Dokument d_i:

$$P(L_j|d_i) = \frac{\prod_{r=1}^{|F|}\left[P(f_r = 0|L_j) * \left(\frac{P(f_r=1|L_j)}{P(f_r=0|L_j)}\right)^{f_{r_{d_i}}}\right] * P(L_j)}{P(d_i)} \tag{2.6}$$

Diese Wahrscheinlichkeit wird als die „A-posteriori-Wahrscheinlichkeit“ für die Klasse L_j für das Dokument d_i bezeichnet, die, im Gegensatz zu den A-priori-Wahrscheinlichkeiten für die Klassen[1], unter dem Einfluss der zur Verfügung stehenden Daten gebildet wurde.

1 Vgl. Mitchell (1997), S. 156.

Nach Weiss u.a. (2005)[1] lässt sich die Formel 2.6 folgendermaßen umformulieren[2]:

$$P(L_j|d_i) = \frac{1}{P(d_i)} \exp\left(\sum_{r=1}^{|F|} w_r * f_{r_{d_i}} + b\right) \text{ mit}$$
$$w_r = \ln\left(\frac{P(f_r = 1|L_j)}{P(f_r = 0|L_j)}\right) \text{ und}$$
$$b = \ln(P(L_j)) + \sum_{r=1}^{|F|} \ln(P(f_r = 0|L_j))$$

unter Berücksichtigung der Formeln 2.4 und 2.5. Also ergibt sich:

$$P(L_j|d_i) = \frac{1}{P(d_i)} \exp\left(\sum_{r=1}^{|F|} \ln\left(\frac{P(f_r = 1|L_j)}{P(f_r = 0|L_j)}\right) * f_{r_{d_i}} + \ln(P(L_j)) + \sum_{r=1}^{|F|} \ln(P(f_r = 0|L_j))\right) \tag{2.7}$$

Es handelt sich hierbei um das so genannte *multivariate Bernoulli Modell*[3]. Für die Klassifizierung von Text eignet sich jedoch ein anderes Modell, das multinomiale, besser, da es die Längen der Dokumente normalisiert.[4] Es ergänzt die Formel um λ, wobei $\lambda > 0$. Dabei handelt es sich um einen Glättungsparameter, der sehr oft auf 1 gesetzt wird.[5] In einigen Fällen ist jedoch ein kleinerer Wert sinnvoller.[6] Das ist in der Implementierung der vorliegenden Arbeit der Fall. Der Glättungsparameter bringt die im Vektor fehlenden Features zur Geltung. In den Experimenten der vorliegenden Arbeit stehen potenziell sehr viele Features zur Verfügung, wovon aber nur ein geringer Anteil pro Dokument vorhanden ist. Um das Ergebnis nicht zu verfälschen, muss dieser Wert sehr klein gewählt werden. Er wird in der vorliegenden Arbeit auf $0,01 * 10^{-95}$ festgelegt. Zudem werden nur die Features explizit in die

1 Vgl. Weiss u.a. (2005), S. 69.
2 Eine Umformung der Formel 2.6 in die Formel 2.7 befindet sich im Anhang A auf Seite 572 f. Die Umformung findet vor allem statt, damit der Logarithmus anstatt des Produkts verwendet werden kann. Das ist für die spätere Implementierung wichtig, da dort sonst evtl. zu kleine Werte berechnet werden müssen, vgl. Manning u.a. (2008), S. 239. Diese sind in der Programmiersprache evtl. nicht mehr darstellbar. Zusätzlich hat das den Effekt, dass der Ausdruck einfacher zu maximieren ist, vgl. Russell u.a. (2003), S. 716. Zulässig ist dies, da der Logarithmus eine streng monoton steigende Funktion ist und daher gilt: Wenn der Logarithmus eines Wertes maximal ist, ist auch der Wert selbst maximal, vgl. Mitchell (1997), S. 166; Manning u.a. (2008), S. 239.
3 Vgl. Weiss u.a. (2005), S. 69.
4 Vgl. Weiss u.a. (2005), S. 69.
5 Vgl. Weiss u.a. (2005), S. 69.
6 Vgl. Weiss u.a. (2005), S. 69.

Berechnung einbezogen, deren Wert gleich 1 ist. Daher sind w_r und b wie folgt definiert[1]:

$$P(L_j|d_i) = \frac{1}{P(d_i)} \exp\left(\sum_{r=1}^{|F|} w_r * f_{r_{d_i}} + b\right) \text{ mit}$$
$$w_r = \ln\left(\frac{\lambda + \left|\left\{\vec{f}_{d_a} \middle| a = 1..|D| \wedge d_a \in L_j \wedge f_{r_{da}} = 1\right\}\right|}{\lambda * |F| + \sum_{r'=1}^{|F|} f_{r'_{d_i}}}\right) \text{ und}$$
$$b = \ln\left(\frac{|L_j|}{|D|}\right)$$

also

$$P(L_j|d_i) = \frac{1}{P(d_i)} \exp\left(\sum_{r=1}^{|F|} \ln\left(\frac{\lambda + \left|\left\{\vec{f}_{d_a} \middle| a = 1..|D| \wedge d_a \in L_j \wedge f_{r_{da}} = 1\right\}\right|}{\lambda * |F| + \sum_{r'=1}^{|F|} f_{r'_{d_i}}}\right) * f_{r_{d_i}} + \ln\left(\frac{|L_j|}{|D|}\right)\right) \tag{2.8}$$

Der Faktor $\frac{1}{P(d_i)}$ kann weggelassen werden, da er das Ergebnis im Hinblick auf die Rangfolge der Klassen für das Dokumente nicht beeinflusst.[2] Insgesamt ergibt sich also für die Berechnung der Bewertung[3], dass ein Dokument d_i zu einer Klasse L_j gehört:

$$B(L_j|d_i) = \exp\left(\sum_{r=1}^{|F|} \ln\left(\frac{\lambda + \left|\left\{\vec{f}_{d_a} \middle| a = 1..|D| \wedge d_a \in L_j \wedge f_{r_{da}} = 1\right\}\right|}{\lambda * |F| + \sum_{r'=1}^{|F|} f_{r'_{d_i}}}\right) * f_{r_{d_i}} + \ln\left(\frac{|L_j|}{|D|}\right)\right) \tag{2.9}$$

1 Vgl. Weiss u.a. (2005), S. 69.

2 Vgl. Weiss u.a. (2005), S. 67; Segaran (2007), S. 125; Mitchell (1997), S. 157; Manning u.a. (2008), S. 245.

3 Durch das Weglassen des Faktors $\frac{1}{P(d_i)}$ handelt es sich nicht mehr um eine Wahrscheinlichkeit. Daher wird in der vorliegenden Arbeit der Terminus Bewertung gewählt.

2.3.3.1.3 Naive-Bayes-Algorithmus zur Klassifizierung von Texten

Zur Klassifizierung von Texten benötigt man die Klasse L_j mit der *maximalen* Bewertung über alle Klassen L, wenn man ein bestimmtes Dokument d_i betrachtet.[1] Formal ausgedrückt bedeutet das[2]:

$$L_{j_{d_i}} = \arg\max_{L_j \in L} \left[\exp\left(\sum_{r=1}^{|F|} \ln\left(\frac{\lambda + \left| \left\{ \vec{f}_{d_a} \,\middle|\, a = 1..|D| \wedge d_a \in L_j \wedge f_{r_{d_a}} = 1 \right\} \right|}{\lambda * |F| + \sum_{r'=1}^{|F|} f_{r'_{d_i}}} \right) * f_{r_{d_i}} + \ln\left(\frac{|L_j|}{|D|} \right) \right) \right] \tag{2.10}$$

In der Anwendung zur Klassifizierung von Texten, um diesen genau eine Klasse zuzuweisen, läuft der Naive-Bayes-Algorithmus folgendermaßen ab[3]:

Algorithmus 1 Naive-Bayes-Algorithmus

Eingabe: Trainingsmenge TR, Klassifikation der Trainingsdaten mit Klassenmenge L, Testmenge TE

Ausgabe: Klassenzuordnung für jedes Dokument aus TE

1: **for all** Klasse $L_j \in L$ **do**
2: Berechne $P(L_j)$[4]und speichere das Ergebnis
3: **for all** Dokument $d_a \in TR$ **do**
4: Berechne $P(d_a|L_j)$ und speichere das Ergebnis
5: **end for**
6: **end for**
7: **while** TE enthält nicht-klassifizierte Dokumente **do**
8: Wähle Dokument $d_i \in TE$

1 Das ist nur der Fall bei der Zuordnung zu *einer* Klasse. Da die durchgeführten Experimente nur diesen Fall berücksichtigen, wird der Algorithmus auch nur für diesen Fall erläutert. Sollen mehrere Klassen zugeordnet werden, so kann man bspw. einen Schwellenwert festlegen. Allen Klassen, deren berechnete Bewertung über dem Schwellenwert liegt, wird dann das betrachtete Dokument zugewiesen. In den Experimenten wird auch die Hierarchie in der Klassifikation nur insofern beachtet, als dass die vorhandenen Vaterklassen ebenfalls hinzugefügt werden, die Algorithmen jedoch nicht für den hierarchischen Fall angepasst werden.

2 Vgl. Manning u.a. (2008), S. 239; Mitchell (1997), S. 157.

3 Der Algorithmus ist hier an die Problemstellung angepasst worden. Ähnliche Algorithmusbeschreibungen befinden sich in Manning u.a. (2008), S. 241; Segaran (2007), S. 123 f. und 126 f. Zhu u.a. (2011), S. 569.

4 Hierbei entspricht $|D|$ $|TR|$.

Algorithmus 1 Naive-Bayes-Algorithmus (Teil 2)

```
    maxRating = 0
    maxClass = „“
    for all Klasse L_j ∈ L do
        result = B(L_j|d_i)
        if result ≥ maxRating then
            maxRating = result
            maxClass = L_j
        end if
    end for
    Weise dem Dokument d_i die Klasse maxClass zu
    Entferne das Dokument d_i aus TE
end while
```

Zunächst werden also die A-priori-Wahrscheinlichkeiten $P(L_j)$ und $P(d_a|L_j)$ für alle Dokumente d_a der Trainingsmenge TR und alle Klassen L der Klassifikation ermittelt und gespeichert, um in späteren Berechnungen darauf zugreifen zu können. Solange nicht-klassifizierte Dokumente in der Testmenge TE enthalten sind, wird das nächste zu klassifizierende Dokument d_i bestimmt. Für dieses muss für alle möglichen Klassen die Bewertung $B(L_j|d_i)$ dafür, dass das Dokument d_i in der Klasse L_j enthalten ist, berechnet werden. Dabei werden jedes Mal, wenn eine größere Bewertung als bisher aufgetreten ist oder nochmals die bisher größte aufgetretene Bewertung berechnet wird, diese Bewertung und die dazugehörige Klasse gespeichert.[1] Abschließend wird dem Dokument die Klasse mit der höchsten Bewertung zugewiesen und mit dem nächsten, noch nicht-klassifizierten, Dokument fortgefahren, sofern noch nicht alle Dokumente der Menge TE klassifiziert wurden.

2.3.3.2 k-Nearest-Neighbour-Algorithmus

Der Algorithmus sagt die Klasse L_j des zu klassifizierenden Dokuments d_i aus der Testmenge TE anhand der k ähnlichsten Dokumente aus der Trainingsmenge TR vorher. Das bedeutet, der Vektor $\vec{f}_{d_i}$ des zu klassifizierenden Dokuments d_i wird mit allen Vektoren $\vec{f}_{d_a}$ der Dokumente d_a in der Trainingsmenge verglichen und seine Ähnlichkeit zu diesen Vektoren berechnet.[2] Diese Annahme, dass die Eigenschaf-

1 In der hier angegegebenen Aufbereitung des Algorithmus wird auf größer oder gleich verglichen. Das bedeutet, wenn es mehrere Klassen für das Dokument gibt, die die gleiche Bewertung aufweisen, so wird nur die letzte dieser Klassen dem Dokument zugewiesen. Betrachtet man den multi-label Fall, so könnten alle Klassen mit der gleichen höchsten Bewertung zugewiesen werden.

2 Vgl. Weiss u.a. (2005), S. 56.

ten eines zu klassifizierenden Dokuments denen der „umliegenden“ Dokumente sehr ähnlich sind, und die eigentliche Ähnlichkeitsberechnung können als Herzstück aller Nächsten-Nachbarn-Methoden angesehen werden.[1] Die Berechnung der Ähnlichkeit kann mit verschiedenen Ähnlichkeitsmaßen durchgeführt werden.[2] In dem hier vorgestellten Algorithmus wird der Cosinus zwischen den Vektoren als Ähnlichkeitsmaß verwendet.
Anschließend werden die k Vektoren bestimmt, zu denen der Vektor des zu klassifizierenden Dokuments am ähnlichsten ist, und die Klasse L_j des zu klassifizierenden Dokuments vorhergesagt. Diese Vorhersage kann auf verschiedene Art erfolgen. Mögliche Arten sind[3]:

1. Zuweisung zur Klasse, die innerhalb der k nächsten Nachbarn am häufigsten auftritt, also den höchsten Anteil an den Klassen der nächsten Nachbarn einnimmt.[4] Hier werden die k nächsten Nachbarn im Hinblick auf ihre Klassen betrachtet und es wird gezählt, wie oft sie zu jeweils einer Klasse gehören. Dann wird ihr jeweiliger Anteil an der Gesamtzahl an nächsten Nachbarn berechnet und die Klasse mit dem höchsten Anteil zugewiesen. Also als Beispiel: $k = 3$, d_1 ist aus Klasse L_1 ebenso wie d_2, d_3 gehört zu Klasse L_2, während L_3 nicht durch einen nächsten Nachbarn vertreten ist. Dann ergibt sich der Anteil für Klasse L_1 im Hinblick auf das zu klassifizierende Dokument d_i mit $\frac{|L_1|}{k} = \frac{2}{3}$, für Klasse L_2 zu $\frac{1}{3}$ und für Klasse L_3 zu 0. Das Dokument d_i wird deshalb der Klasse L_1 zugewiesen.

2. Zuweisung zur Klasse des Dokuments d_a mit dem höchsten Cosinuswert zwischen dem Vektor von d_a und dem von d_i.

1 Vgl. Weiss u.a. (2005), S. 88; Russell u.a. (2003), S. 733.
2 Für einen Überblick über Ähnlichkeitsmaße siehe Kapitel 2.2. Vgl. bspw. auch Russell u.a. (2003), S. 734 f.
3 Da in den durchgeführten Experimenten eine Single-label-Klassifizierung erfolgt, wird an dieser Stelle nur auf Möglichkeiten eingegangen, die ebenfalls jedem Dokument genau eine Klasse zuordnen. Um eine Multi-label-Klassifizierung durchzuführen, existieren ebenfalls verschiedene Möglichkeiten. So berechnen bspw. Lewis u.a. (2004) eine Punktzahl für alle Klassen der k nächsten Nachbarn und legen einen Klassenschwellenwert fest. Für jede Punktzahl, die diesen Schwellenwert überschreitet, wird die Klasse dem entsprechenden Dokument zugeordnet, vgl. Lewis u.a. (2004), S. 383. Eine Anpassung des k-Nearest-Neighbour-Algorithmus für den Multi-label-Fall beschreiben bspw. Zhang u.a. (2005) und Younes (2011). Die Verwendung einer Hierarchie wird in den Experimenten nur insoweit betrachtet, als dass alle Vaterklassen ebenfalls einem Dokument zugeordnet werden.
4 Vgl. Weiss u.a. (2005), S. 55; Russell u.a. (2003), S. 735; Manning u.a. (2008), S. 274. Treten mehrere Klassen mit der gleichen größten Häufigkeit auf, so muss eine Strategie gefunden werden, wie damit umzugehen ist. Das könnte eine randomisierte Auswahl einer der Klassen sein.

Hier würde man im engeren Sinne nur den nächsten Nachbarn betrachten, also $k = 1$ setzen. Das ist nicht sehr vorteilhaft, insbesondere bei einer großen Datenmenge.[1]

3. Zuweisung zur Klasse mit der höchsten Punktzahl (englisch: *score*), basierend auf der Cosinusähnlichkeit. Die Punktzahl wird wie folgt festgelegt[2]:

$$\text{score}(L_j, d_i) = \sum_{a=1}^{k} I_{L_j}(d_a) \cos(\vec{f}_{d_a}, \vec{f}_{d_i}), \text{ wobei}$$

$$I_{L_j}(d_a) = \begin{cases} 1 & \text{wenn } d_a \in L_j \\ 0 & \text{sonst} \end{cases}$$

Die Klasse mit der höchsten Punktzahl wird dann d_i zugewiesen.[3]

Als Algorithmus zusammengefasst ergibt sich[4]:

Algorithmus 2 k-Nearest-Neighbour-Algorithmus

Eingabe: Trainingsmenge TR, Klassifikation der Trainingsdaten mit Klassenmenge L, Testmenge TE

Ausgabe: Klassenzuordnung für jedes Dokument aus TE

1: **for all** Dokument $d_i \in TE$ **do**
2: **for all** Dokument $d_a \in TR$ **do**
3: Berechne die Ähnlichkeit zwischen d_a und d_i und speichere das Ergebnis
4: **end for**
5: Sortiere alle d_a absteigend nach der berechneten Ähnlichkeit mit d_i
6: Weise d_i eine Klasse aufgrund der k ähnlichsten Dokumente und der Art der Zuweisung zu
7: **end for**

Ein großer Vorteil dieses Algorithmus ist, dass nahezu kein Trainingsaufwand nötig ist, weil lediglich die Trainingsbeispiele gespeichert werden müssen, ohne weitere

1 Vgl. Weiss u.a. (2005), S. 55, 88. Manning u.a. (2008), S. 274, begründen das damit, dass dieses nächste Dokument, auf dem allein die Klassifizierung des Testdokuments basiert, durchaus fehlerhaft klassifiziert worden sein kann.
2 Vgl. Manning u.a. (2008), S. 275. Die Punktzahl kann auch anders festgelegt werden, vgl. bspw. Lewis u.a. (2004), S. 383.
3 Hier sinkt die Wahrscheinlichkeit, dass es mehrere Klassen mit der gleichen Anzahl an Nachbarn gibt, vgl. Manning u.a. (2008), S. 275.
4 Vgl. Weiss u.a. (2005), S. 55.

Berechnungen durchzuführen.[1] Laut Segaran (2007)[2] hat die Methode noch einen weiteren entscheidenden Vorteil: sie ist für den Menschen nachvollziehbar. Das bedeutet, die Klassifizierung wird automatisch durchgeführt, jedoch kann man die Entscheidungen des Systems nachvollziehen, wenn man sich die nächsten Nachbarn ausgeben lässt. Trotzdem können mit ihrer Hilfe komplexe Entscheidungen durchgeführt werden.

Dazu kommt, dass eine Erweiterung der Trainingsdaten keine Neuberechnung der Ähnlichkeiten für die Dokumente aus der alten Trainingsmenge nach sich zieht.[3] Das führt jedoch dazu, dass für jedes zu klassifizierende Dokument eine andere Zielfunktion geschätzt wird, da lediglich der lokale Bereich um das zu klassifizierende Dokument betrachtet wird und nicht alle zur Verfügung stehenden Trainingsdokumente. Der Alogithmus schätzt also viele verschiedene Zielfunktionen und nicht eine für alle zukünftigen zu klassifizierenden Dokumente.[4]

Hier zeichnet sich bereits ein Nachteil dieses Algorithmus ab: Der gesamte Aufwand zur Durchführung der Methode liegt im Klassifizierungsschritt. D.h., während der Trainingsaufwand sehr gering ist, muss für jedes zu klassifizierende Dokument erneut eine Berechnung durchgeführt werden, ohne auf vorher getätigte Berechnungen zurückgreifen zu können.[5] Ein weiterer Nachteil, mit dem bei der Klassifizierung von Texten viele Methoden behaftet sind, ist der so genannte „curse of dimensionality“[6], womit gemeint ist, dass alle Features bei der Ähnlichkeitsberechnung in Betracht gezogen werden, auch wenn einige oder viele nichts dazu beitragen, die Dokumente korrekt zu klassifizieren. Diese „überflüssigen“ Features können die Ähnlichkeitsberechnung verzerren und zu falschen Ergebnissen führen.[7]

Eine nicht zu vergessende Schwierigkeit beim Algorithmus beinhaltet das Festlegen des Wertes für k. Es handelt sich bei diesem Parameter um eine Möglichkeit, die „Nachbarschaft“ der Dokumente zu bestimmen, ohne die wirkliche Verteilung der Dokumentvektoren im Raum zu kennen. D.h., durch das Belegen von k mit einem

1 Vgl. Weiss u.a. (2005), S. 58; Mitchell (1997), S. 230. Eine Ausnahme bildet der Fall, wenn k noch bestimmt werden müsste. Hier wird jedoch davon ausgegangen, dass der Wert bereits zuvor festgelegt wurde.

2 Vgl. Segaran (2007), S. 296.

3 Vgl. Segaran (2007), S. 171.

4 Vgl. Mitchell (1997), S. 231. Durch eine Gewichtung der Trainingsbeispiele, die abhängig von der Ähnlichkeit mit dem zu klassifizierenden Dokument ist, könnten auch alle Trainingsbeispiele einbezogen werden und der Algorithmus so von einem lokalen Algorithmus zu einem globalen werden, da die unähnlichen Trainingsbeispiele nur ein geringes Gewicht erhalten würden, vgl. Mitchell (1997), S. 233 f.; Segaran (2007), S. 172-176.

5 Vgl. Mitchell (1997), S. 231.

6 Mitchell (1997), S. 235.

7 Um dieses Problem in Bezug auf den k-Nearest-Neighbour-Algorithmus zu lösen, existieren verschiedene Ansätze. Einige davon werden in Mitchell (1997), S. 235 f., und Segaran (2007), S. 178-188, vorgestellt.

„guten“ Wert wird der Bereich der betrachteten Nachbarn nicht zu groß oder zu klein festgelegt, da nur die Anzahl der Nachbarn entscheidend ist, nicht aber, wo genau im Raum diese liegen. So kann es bei zerstreuten Daten sehr weit entfernte Dokumente geben, die jedoch trotzdem betrachtet werden, oder bei dichten Daten Dokumente, die sehr eng am zu klassifizierenden liegen.[1]
Um k festzulegen, existieren verschiedene Möglichkeiten:

1. Schätzen von k durch Experimente[2]
   Dabei werden verschiedene Experimente mit jeweils unterschiedlichen Werten für k durchgeführt, die Ergebnisqualität wird gemessen und schließlich wird der Wert für k gewählt, der in den Experimenten am besten abgeschnitten hat.

2. Schätzen von k durch Kreuzvalidierung[3]
   Kreuzvalidierung bedeutet, dass die Trainingsdaten in mehrere Teilmengen geteilt werden. Jeweils eine davon wird nicht zum Training des Klassifizierers genutzt. Mit den anderen wird der Klassifizierer trainiert, wobei jeweils ein anderer Wert für k verwendet wird. Anschließend wird der jeweilige trainierte Klassifizierer mit dem jeweiligen k auf die bisher nicht verwendete Menge an Trainingsdaten angewendet. Sie dient als Testdatenmenge. Anschließend werden die erreichten Ergebnisse auf ihre Qualität hin untersucht und es wird der Wert für k gewählt, mit dem das beste Ergebnis erzielt wurde.

3. Manuelle Festlegung von k für verschiedene Datenmengen[4]
   Dafür wird eine Expertenmeinung benötigt von jemandem, der die Daten sehr genau kennt.

2.3.3.3 Entscheidungsbaum-Algorithmus

Die Idee, Entscheidungsbäume (englisch: *decision trees*) aufgrund vorhandener Trainingsbeispiele zu erzeugen, stammt von Hovelant und Hunt[5]. Die Grundidee ist die Aufteilung der Menge der Trainingsdaten in kleinere Teilmengen so lange, bis sie eine Klasse symbolisieren, der unbekannte Dokumente anhand ihrer Features zugeordnet werden können. Diese Aufteilung erfolgt in Baumform, wobei jeder Knoten den Test eines Features im Hinblick auf seinen Wert darstellt.[6] D.h., die Features bilden die

1 Vgl. Russell u.a. (2003), S. 733.
2 Vgl. Weiss u.a. (2005), S. 56, 88; Manning u.a. (2008), S. 274.
3 Vgl. Manning u.a. (2008), S. 274.
4 Vgl. Segaran (2007), S. 170.
5 Vgl. Quinlan (1993), S. 17.
6 Vgl. Quinlan (1993), S. 17 f.

Knoten des Entscheidungsbaums. Entscheidungsbäume werden häufig bei der Bearbeitung von Klassifizierungsproblemen eingesetzt. So finden Entscheidungsbäume bspw. Anwendung bei der medizinischen Diagnose und der Risikobetrachtung bei der Kreditvergabe.[1]
Der Entscheidungsbaum wird mit Hilfe der Dokumente der Trainingsmenge TR aufgebaut. Dabei bilden ausgewählte Features dieser Dokumente die Knoten und jeder Zweig, der zum nächsten Kindknoten im Baum führt, beschreibt einen möglichen Wert des Features.[2] Durch die Repräsentation der Dokumente als binäre Vektoren handelt es sich bei den Werten nur um zwei: „Feature ist vorhanden“ oder „Feature ist nicht vorhanden“. Das Feature an der Wurzel ist dasjenige, das die Trainingsbeispiele am Besten aufteilt. Seine Kindknoten sind dann die Features, die für eine weitere Aufteilung sorgen, so lange, bis an den Blattknoten die letztendliche Entscheidung für die Klasse fallen kann.
Ein Dokument der Testmenge TE wird klassifiziert, indem der Baum von der Wurzel an abwärts durchlaufen wird. Dabei wird jeweils das entsprechende Feature im Vektor des zu klassifizierenden Dokuments betrachtet und je nach An- oder Abwesenheit die Richtung des Abstiegs gewählt. Das geschieht, bis ein Blattknoten erreicht wurde und dem zu klassifizierenden Dokument die dort festgelegte Klasse zugeordnet wird.[3]
Der Aufbau des Entscheidungsbaums in dieser Arbeit basiert auf dem ID3-Algorithmus[4], wandelt diesen jedoch etwas ab[5]. Der Aufbau erfolgt rekursiv. Zunächst wird entschieden, welches Feature an der Wurzel getestet werden soll, dann wird für die beiden möglichen Werte dieses Features eine Verzweigung angelegt und das gleiche Verfahren rekursiv für jede Verzweigung angewendet. Um die Entscheidung zu treffen, welches Feature sich am besten als Wurzelknoten eignet, benötigt man ein quantitatives Kriterium zum Messen der Güte des Features. Das Feature wird daraufhin getestet, wie gut es alleine, ohne weitere Features in Betracht zu ziehen, die vorhandenen Trainingsbeispiele bereits klassifizieren würde.[6] Dieses statistische Kriterium nennt man *Information Gain*.

1 Vgl. Mitchell (1997), S. 52.
2 Vgl. Mitchell (1997), S. 52 f.; Russell u.a. (2003), S. 653.
3 Vgl. Mitchell (1997), S. 52 f.; Russell u.a. (2003), S. 653; Segaran (2007), S. 152.
4 Vgl. Quinlan (1986), insbesondere S. 87-92. Eine Weiterentwicklung des ID3-Algorithmus ist der C4.5-Algorithmus, ebenfalls von Quinlan entwickelt, vgl. Quinlan (1993), S. 115-287, insbesondere S. 122-125. Es existieren noch weitere ähnliche Algorithmen zum Aufbau von Entscheidungsbäumen, bspw. CART (Classification and Regression Trees), vgl. Breiman u.a. (1993), insbesondere S. 27-36.
5 Siehe Algorithmus 3 auf S. 55 f.
6 Vgl. Mitchell (1997), S. 55; Russell u.a. (2003), S. 656; Segaran (2007), S. 145.

Zur Berechnung des Information Gain ist die so genannte Entropie[1] wichtig. Sei S die betrachtete Teilmenge der Trainingsdaten beim Aufbau des Baums, $|S_{L_j}|$ die Anzahl der Trainingsbeispiele der Teilmenge S mit Klassenzuordnung L_j und $|L_j|$ die Anzahl der Trainingsdaten mit zugeordneter Klasse L_j über alle Trainingsdaten, dann lautet die allgemeine Definition der Entropie[2]:

$$\text{Entropie}(S) = \sum_{j=1}^{|L|} -p_j * \log_2(p_j) \text{ mit}$$
$$p_j = \frac{|S_{L_j}|}{|L_j|}$$

Dabei gilt zusätzlich: $0 * \log_2(0) = 0$.
Der Information Gain misst die erwartete Reduktion der Entropie. Das bedeutet, wenn die Entropie die Unreinheit von Trainingsdaten misst, dann misst der Information Gain die Effektivität der Klassifizierung von Trainingsdaten durch ein Feature.[3] Der Information Gain ist die erwartete Reduktion der Entropie der anderen Features dadurch, dass man den Wert eines Features kennt. Man kann auch sagen, er drückt die Zahl der Bits aus, die bei der Kodierung eines beliebigen Trainingsbeispiels aus S gespart werden können. Eine weitere Formulierung wäre, dass berechnet wird, wieviel Information noch nötig ist, nachdem dieses Feature mit einem Wert belegt wurde.[4]

1 Eine andere Bezeichnung dafür ist die „Menge an Information“ für ein Feature, vgl. Russell u.a. (2003), S. 659. Die Entropie oder das Maß des Informationsgehalts stammt aus der Informationstheorie und spezifiziert, wie stark eine Information komprimiert werden kann, d.h., wie viele Bits für die Übermittlung benötigt werden, vgl. Mitchell (1997), S. 57; Russell u.a. (2003), S. 659; Shannon u.a. (1963), S. 9, 12, 15. Eine weitere Interpretation des Maßes ist die als Unreinheit der Trainingsdaten, also wie sehr die Trainingsbeispiele an einem Knoten, bezogen auf ihre Klassen, gemischt sind, vgl. Segaran (2007), S. 148; Mitchell (1997), S. 57; Shannon u.a. (1963), S. 12.

2 Vgl. Mitchell (1997), S. 57; Russell u.a. (2003), S. 659; Shannon u.a. (1963), S. 50.

3 Vgl. Mitchell (1997), S. 57.

4 Vgl. Russell u.a. (2003), S. 660. Die Aussage ist, dass jedes mit einem Wert belegte Feature die Trainingsbeispiele aufgrund ihrer Werte für dieses Feature aufteilt. Die Beispiele selbst sind wiederum Klassen zugeordnet. Um nun die jeweilige Klasse für ein Dokument zu bestimmen, ist wieder zusätzliche Information in Form von Bits notwendig. Der Information Gain ist die Differenz zwischen dem ursprünglichen Informationsbedarf und dem neu bestimmten. Ist diese Differenz möglichst hoch, so ist der Nutzen, der aus dem Attribut gezogen werden kann, möglichst hoch und es wird ausgewählt.

Er wird wie folgt berechnet[1]:

$$\text{Gain}(S, f_r) = \text{Entropie}(S) - \sum_{v=1}^{|V_{fr}|} \frac{|S_v|}{|S|} * \text{Entropie}(S_v), \text{ wobei}$$

$|V_{f_r}|$ die Anzahl der Werte des Features f_r ist,

$\frac{|S_v|}{|S|}$ die Anzahl der Trainingsbeispiele aus S mit Wert v sind, dividiert durch die Anzahl der Trainingsbeispiele in S,

$\text{Entropie}(S_v)$ die Entropie der Menge der Trainingsbeispiele mit dem Wert v für das Feature f_r ist.

Anhand des Information Gain kann zu Beginn des Baumaufbaus die Entscheidung darüber getroffen werden, welches Feature am besten dazu geeignet ist, die Wurzel des Entscheidungsbaums zu bilden, und in den nachfolgenden rekursiven Schritten beim Baumaufbau, welches Feature als nächster Knoten zu wählen ist. D.h., der Wurzelknoten wird aufgrund des maximalen Information Gain für das entsprechende Feature festgelegt und die jeweiligen Trainingsbeispiele mit den Werten des Features den möglichen Verzweigungen zugeordnet. Danach folgt eine Wiederholung des Prozesses, so dass das nächste Feature als ein Knoten des Baums festgelegt werden kann.[2]
Dieser Prozess wird so lange weitergeführt, bis an jedem neuen Knoten eine von zwei Bedingungen erfüllt ist[3]:

(1) Jedes Feature ist einmal auf dem Pfad von der Wurzel bis zum Blattknoten enthalten.

(2) Die Trainingsbeispiele, die zu diesem Knoten gehören, sind alle derselben Klasse L_j zugeordnet.

1 Vgl. Mitchell (1997), S. 58; Russell u.a. (2003), S. 660.
2 Vgl. Mitchell (1997), S. 55; Segaran (2007), S. 149.
3 Vgl. Mitchell (1997), S. 60.

Als in dieser Arbeit verwendeter Algorithmus zum Aufbau eines Decision Trees ergibt sich der folgende[1]:

Algorithmus 3 Decision Tree aufbauen

Eingabe: Trainingsmenge TR, Featuremenge F

Ausgabe: Decision Tree DT, der alle Dokumente aus TR richtig klassifiziert

1: **if** $TR == \emptyset$ **then**
2: leeren Baum zurückgeben
3: **end if**
4: Erzeuge *Root*, den Wurzelknoten des (Teil-)Baums
5: **if** alle Dokumente in TR gehören zur gleichen Klasse **then**
6: *Root* als Blattknoten setzen
7: *Root* Klasse der Dokumente aus TR zuweisen
8: *Root* zurückgeben
9: **end if**
10: **if** alle Features in F haben den gleichen Wert **then**
11: *Root* als Blattknoten setzen
12: *Root* Klasse, die mit der größten Häufigkeit in den Dokumenten aus TR auftritt, zuweisen
13: *Root* zurückgeben
14: **end if**
15: Finde das Feature mit höchstem Information Gain, das ist $maxGainF$ ▷ An dieser Stelle ist klar, dass es sich bei *Root* nicht um einen Blattknoten handelt
16: Entferne Feature $maxGainF$ aus der Featuremenge F, die Teilmenge ist $newF$
17: Schränke TR auf die Dokumente ein, die 0 als Belegung für $maxGainF$ haben, die Teilmenge ist S_0
18: $Root.Kind0$ = DECISION TREE AUFBAUEN(S_0, $newF$)▷ Erzeuge den Teilbaum für den Wert 0
19: Schränke TR auf die Dokumente ein, die 1 als Belegung für $maxGainF$ haben, die Teilmenge ist S_1
20: $Root.Kind1$ = DECISION TREE AUFBAUEN(S_1, $newF$)▷ Erzeuge den Teilbaum für den Wert 1
21: *Root* zurückgeben

1 Es handelt sich um eine Abwandlung des ID3-Algorithmus nach Mitchell (1997), S. 56 und ähnelt Moore (2011), Folie 45.

2.3.3.4 Support-Vector-Machine-Algorithmus

Ein weiterer Algorithmus, der zum Klassifizieren verwendet werden kann, ist die Support Vector Machine (SVM). Sie wurde von Vapnik (1995)[1] und Cortes u.a. (1995)[2] vorgestellt und basiert auf der Idee, Daten linear klassifizieren zu können. Ein linearer Klassifizierer ist im geometrischen Sinne ein Klassifizierer, der die vorhandenen Daten durch ein geometrisches Objekt als Trenner in zwei Klassen teilen kann. Segaran (2007) beschreibt den Vorgang wie folgt[3]:

- Man berechnet den Durchschnitt der Daten pro Klasse und erhält so das Zentrum der jeweiligen Klasse.[4]
- Die Klassenzentren und ihre dazugehörenden Daten können durch eine Gerade geteilt werden.
- Für neue Daten wird berechnet, welchem Zentrum sie am nächsten liegen, und sie werden der Klasse zugeordnet, deren Zentrum sie am nächsten liegen.

Entscheidend bei dieser Vorgehensweise ist die Berechnung der Nähe. Diese erfolgt bei Segaran (2007) mit Hilfe des Skalarprodukts zweier Vektoren.[5]
In der Verallgemeinerung wird die Gerade, die die Daten teilt, als *Hyperebene* (engl. *hyperplane*) bezeichnet.[6] Geht man von einer Geraden aus, die die Daten teilt, so ist sie definiert durch: $\vec{w} * \vec{x} + b = 0$[7], wobei $\vec{w}$ und b die beiden Parameter sind, die aus den Daten erlernt werden müssen. Die Daten oder besser gesagt, die Datenvektoren, werden durch $\vec{x}$ dargestellt. In der geometrischen Interpretation verschiebt b die Gerade parallel zu sich selbst und $\vec{w}$ ist die Steigung, also die Richtung des Vektors, der senkrecht zur Geraden ist.[8] Daten, die auf diese Weise korrekt klassifiziert werden können, werden *linear separierbar* genannt.[9]

1 Vapnik (1995) zitiert nach Joachims (1998), S. 137.
2 Vgl. Cortes u.a. (1995), S. 274-286.
3 Vgl. Segaran (2007), S. 202. Beschrieben wird der Fall einer binären Klassifizierung.
4 Es wird davon ausgegangen, dass die Daten als Vektoren dargestellt sind, so dass der Durchschnitt dieser Datenvektoren gebildet wird.
5 Beim Skalarprodukt handelt es sich um einen Skalar zweier Vektoren. Man multipliziert jeweils die korrespondierenden Vektorkomponenten und addiert schließlich alle Werte, vgl. Segaran (2007), S. 203.
6 Vgl. Cristianini u.a. (2002), S. 10. Eine solche Hyperebene ist eine Generalisierung einer Ebene im dreidimensionalen Raum. Das bedeutet, in einem n-dimensionalen Raum hat die Hyperebene $n - 1$ Dimensionen. Zur Veranschaulichung: in einem 1-dimensionalen Raum wäre die Hyperebene ein Punkt, im 2-dimensionalen Raum eine Gerade und im 3-dimensionalen Raum eine Ebene, vgl. Noble (2006), S. 1565.
7 Cristianini u.a. (2002), S. 9 f. Die Notation wurde durch die Verfasserin an die in der vorliegenden Arbeit verwendete angepasst.
8 Vgl. Cristianini u.a. (2002), S. 10.
9 Vgl. bspw. Cristianini u.a. (2002), S. 11.

Für die Berechnung der Hyperebene sind nur wenige Datenvektoren nötig.[1] Bei diesen handelt es sich um die Datenvektoren, die den geringsten Abstand zur Hyperebene aufweisen. Sie werden *Stützvektoren* (engl. *support vectors*) genannt.[2] Daher sind in $\vec{x}$ nicht alle $\vec{f}$ der Trainingsmenge für eine Klassifizierung enthalten, sondern nur die davon benötigten Stützvektoren.
In der Praxis trifft man selten auf linear separierbare Daten.[3] Trotzdem können hier lineare Klassifizierer verwendet werden. Zuvor muss jedoch eine Transformation der Ursprungsdaten in einen so genannten Merkmalsraum (*feature space*)[4] erfolgen. Dieser Raum hat die Eigenschaft, mehr Dimensionen als der ursprüngliche Vektorraum zu besitzen. Aufgrund dieser zusätzlichen Dimensionen können die Daten im Merkmalsraum linear getrennt werden.[5] Zur Verdeutlichung folgt Abbildung 2.3 auf S. 58, die links geometrisch die ursprünglichen Daten als Punkte abbildet, die nicht linear getrennt werden können, und rechts die transformierten Daten, die linear separierbar sind. Die Abbildung ist Russell u.a. (2003)[6] entnommen.
Diese Transformation und die damit verbundene lineare Separierbarkeit von Daten lässt sich generalisieren: Alle Daten sind linear separierbar, wenn sie in einen Merkmalsraum mit genügend Dimensionen transformiert werden.[7] Insgesamt läuft das Klassifizieren von Daten, die nicht linear separierbar sind, mit linearen Klassifizierern in zwei Schritten ab[8]:

1. Transformation der Daten in einen Merkmalsraum mit mehr Dimensionen,

2. Klassifizieren der transformierten Daten mit einem linearen Klassifizierer, der im Merkmalsraum arbeitet.

Als problematisch kann sich der erste Schritt erweisen, da die Transformation der Daten in der Praxis sehr aufwändig sein kann[9] und die Gefahr einer *Überanpassung*

1 Vgl. Segaran (2007), S. 216; Cortes u.a. (1995), S. 275.
2 Vgl. Cristianini u.a. (2002), S. 97.
3 In der vorliegenden Arbeit wird in den Experimenten vereinfachend davon ausgegangen, dass die Daten linear separierbar sind. Daher wird mit Hilfe eines linearen Support-Vector-Machine-Algorithmus klassifiziert. Der Vollständigkeit halber wird im Rest des Kapitels auch kurz auf den Fall eingegangen, dass die Daten nicht linear separierbar sind. Für eine ausführlichere Darstellung auch der mathematischen Hintergründe sei jedoch jeweils auf die an den entsprechenden Stellen genannte Literatur verwiesen.
4 Dieser Merkmalsraum hat nichts mit den Merkmalen oder Features, die aus den natürlichsprachlichen Texten stammen, zu tun. Es handelt sich lediglich um eine Bezeichnung für einen geometrischen Raum, in den die Datenvektoren aus dem Vektorraum transformiert werden.
5 Vgl. bspw. Segaran (2007), S. 211; Russell u.a. (2003), S. 749; Cristianini u.a. (2002), S. 27.
6 Russell u.a. (2003), S. 750.
7 Vgl. Russell u.a. (2003), S. 749.
8 Vgl. Cristianini u.a. (2002), S. 30.
9 Vgl. Segaran (2007), S. 211.

(engl. *overfitting*)[1] besteht. Jedoch ist es möglich, die Probleme zu umgehen und beide der oben genannten Schritte in einem Schritt durchzuführen[2]. Dafür benutzt man den so genannten „kernel trick“[3].

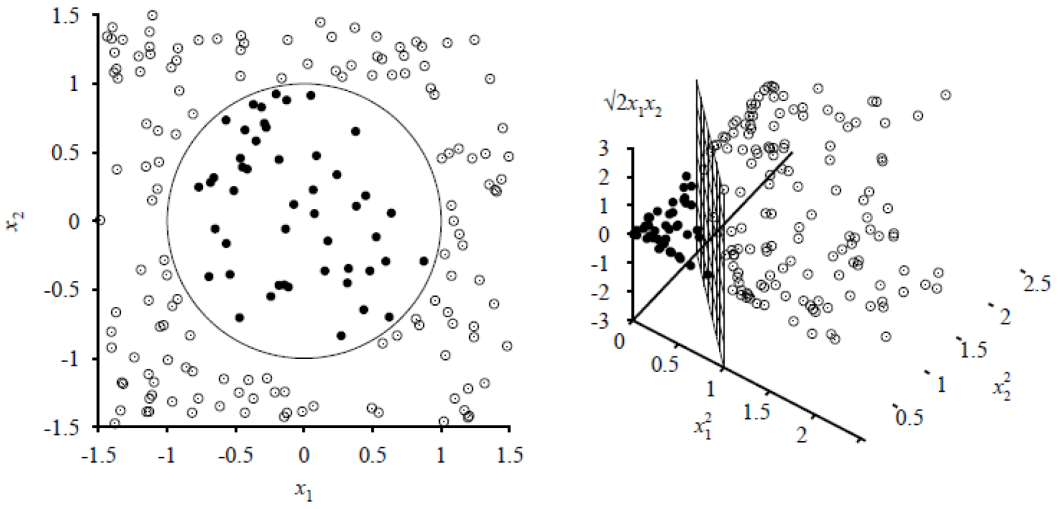


Abbildung 2.3: Linear nicht separierbare Daten und transformierte linear separierbare Daten

Dieser Trick basiert darauf, die Transformation der Daten nicht explizit durchzuführen, sondern durch eine Funktion zu ersetzen, die ein Ergebnis zurückliefert, welches dem Ergebnis entspricht, das die ursprüngliche Berechnung zurückliefern würde, wenn die Daten zuvor transformiert worden wären.[4] Bemerkenswert ist, dass man einen Kernel verwenden kann, ohne das darunterliegende Mapping zu kennen.[5] Es handelt sich laut Cristianini u.a. (2002) um einen „attractive computational shortcut“[6]. Dieser Vorteil bezieht sich nicht nur auf die Verkürzung bei der Berechnung,

1 Vgl. Russell u.a. (2003), S. 749. Eine Überanpassung ist laut Hawkins (2004), S. 1 f., entweder ein zu flexibles Modell oder ein Modell, das nicht benötigte Komponenten enthält. Übertragen auf Support Vector Machines oder auch Decision Trees bedeutet das, dass ein Modell der Trainingsdaten erstellt wird, das diese zwar bestmöglich klassifiziert. Dieses Modell ist jedoch so speziell auf die Trainingsdaten zugeschnitten, dass es nicht mehr generalisieren kann, also unbekannte Daten schlechter klassifiziert als ein anderes Modell, das die Trainingsdaten mit Fehlern klassifiziert. Eine allgemeinere Definition von Überanpassung ist bei Mitchell (1997), S. 67, zu finden. Er beschreibt auch ausführlich die Überanpassung bei Decision Trees und zeigt Möglichkeiten auf, wie diese verringert werden kann, vgl. Mitchell (1997), S. 66-72.
2 Vgl. Cristianini u.a. (2002), S. 30.
3 Segaran (2007), S. 212.
4 Vgl. Segaran (2007), S. 212.
5 Vgl. Cristianini u.a. (2002), S. 31.
6 Cristianini u.a. (2002), S. 32.

indem die explizite Transformation der Daten entfällt, sondern auch auf die Möglichkeit, durch die Verwendung unterschiedlicher Kernelfunktionen verschiedene nichtlineare Klassifizierer erzeugen zu können, ohne die Berechnungsvorschrift gravierend ändern zu müssen.[1] Bekannte mögliche Kernelfunktionen sind, neben dem linearen Kernel, der polynomiale, eine radial basis function und ein neuronales Netzwerk.

Kernel-Maschinen[2] finden den optimalen linearen Trenner.[3] Dabei handelt es sich um die Hyperebene mit dem größten *Rand* (engl. *margin*) zwischen den Klassen, also den Datenvektoren und der Hyperebene. Der Rand zwischen einem Datenvektor und der trennenden Hyperebene ist der Abstand des Datenvektors von der Hyperebene.[4] Der größte Rand zwischen allen Datenvektoren einer Trainingsmenge und der Hyperebene entspricht dem maximalen geometrischen Rand über alle möglichen Hyperebenen[5] und wird *maximum margin hyperplane* genannt.[6] Genau diese gilt es zu finden, wobei zu beachten ist, dass die Hyperebene nicht nur den größten Abstand von allen Trainingsbeispielen, sondern auch von den Testbeispielen haben soll.[7] Um das zu erreichen, muss, basierend auf den Trainingsbeispielen, ein quadratisches Optimierungsproblem gelöst werden.[8] Die Daten gehen dabei nur in Form des Skalarprodukts ein, es kann also eine Kernelfunktion verwendet werden.[9]

Aus diesen Bestandteilen, also den support vectors und der kernel machine, setzt sich der Name des Algorithmus zusammen.[10]

Bei Verwendung der maximum margin hyperplane kann es zu einer Überanpassung kommen.[11] Sind die Trainingsbeispiele zuvor manuell falsch klassifiziert worden oder weisen sie andere Fehler auf, was in der Praxis häufig vorkommen kann, so wird eine SVM erstellt, die Testbeispiele nicht mehr korrekt klassifiziert. Aus diesem Grund

1 Vgl. Bennett u.a. (2000), S. 5; Cortes u.a. (1995), S. 284.
2 Hier wird von der Verfasserin der im Englischen gebräuchliche Ausdruck *machine* in das deutsche *Maschine* übersetzt. Es handelt sich nicht um einen aus Bauteilen bestehenden Gegenstand, sondern um eine Methode aus dem maschinellen Lernen. Daher stammt der Ausdruck *Maschine* oder *machine* im Englischen. Eine andere mögliche Bezeichnung wäre *Automat*.
3 Vgl. Russell u.a. (2003), S. 749.
4 Vgl. Cristianini u.a. (2002), S. 12.
5 Es gibt sehr viele Hyperebenen, die die Daten korrekt klassifizieren würden, jedoch nur eine maximum margin Hyperebene, vgl. Bennett u.a. (2000), S. 1; Noble (2006), 1565 f.
6 Vgl. Cristianini u.a. (2002), S. 12.
7 Vgl. Joachims (1999), S. 203.
8 Vgl. Russell u.a. (2003), S. 749.
9 Vgl. Russell u.a. (2003), S. 751.
10 Da dieser Algorithmus von den Klassifizierungsalgorithmen der einzige ist, der nicht von der Verfasserin implementiert wurde, sondern eine Software benutzt wurde, wird hier auf die Beschreibung des Algorithmus verzichtet. An späterer Stelle dieser Arbeit, siehe Kapitel 4.1.9.2.4, wird die verwendete Software erläutert. Beispiele für einen SVM-Algorithmus befinden sich in Bennett u.a. (2000), S. 5; Cortes u.a. (1995), S. 280.
11 Vgl. Cristianini u.a. (2002), S. 103.

kann eine so genannte *soft margin* SVM erstellt werden.[1] Diese erlaubt einige wenige so genannte *Ausreißer* beim Klassifizieren, also Datenvektoren, die falsch klassifiziert werden. Die Bedingungen für den Rand werden durch so genannte *Schlupfvariablen* (engl. *slack variables*) aufgeweicht und erlauben so auch Datenvektoren falsch zu klassifizieren.[2] Um zu viele Fehler beim Klassifizieren zu vermeiden, wird ein zusätzlicher Parameter C eingeführt, der den Zielkonflikt zwischen den Verletzungen der richtigen Klassenzuordnung und der Größe des Randes festlegt.[3]

2.4 Clustern

2.4.1 Darstellung des Clusterns

Clustern[4] im Allgemeinen bedeutet das Erkennen und Visualisieren von Gruppen („Clustern“) in bestimmten Kontexten, wie z.B. von Ideen, Menschen oder Dingen, die sich auf eine festzulegende Art ähnlich sind.[5]
In dieser Arbeit wird nicht die manuelle Form der Gruppenbildung betrachtet, sondern die automatisierte, von Computern durchgeführte Form. Zusätzlich werden hier natürlichsprachliche Texte, also Dokumente, betrachtet. Als Clustern im Kontext von Dokumenten bezeichnet man die Gruppierung dieser Dokumente in Untermengen, so genannte *Cluster*. Hierbei soll gelten, dass sich Dokumente innerhalb eines Clusters sehr ähnlich sind und Dokumente aus verschiedenen Clustern sehr unähnlich.[6] Die Ähnlichkeit wird aufgrund des Inhalts der Dokumente bestimmt.
Die Menge der Dokumente sei $D = \left\{d_1, d_2, d_3, ..., d_{|D|}\right\}$ und die der zu erzeugenden Cluster $\Omega = \{\omega_1, \omega_2, ..., \omega_c\}$. Es wird eine Zuordnung gesucht, die jedem Cluster die Dokumente so zuordnet, dass eine Zielfunktion γ maximiert oder minimiert wird.[7] Die Zielfunktion ist die folgende:

$$\gamma : D \rightarrow \{\omega_1, ..., \omega_c\}.$$

1 Vgl. Noble (2006), S. 1566; Cortes u.a. (1995), S. 281.
2 Vgl. Cristianini u.a. (2002), S. 15 f., 103 f.
3 Vgl. Noble (2006), S. 1566; Cortes u.a. (1995), S. 286.
4 Als *Clustern* wird in der vorliegenden Arbeit der Vorgang der Erstellung von Clustern verstanden. *Clustering* bezeichnet das Ergebnis des Clusterns, also die Menge der entstandenen Cluster, wobei im Englischen der Begriff *clustering* auch für den Vorgang der Ereugung von Clustern verwendet wird. In der vorliegenden Arbeit wird nur dann *clustering* für den Vorgang der Erzeugung von Clustern verwendet, wenn es sich um einen feststehenden englischen Begriff handelt.
5 Vgl. Segaran (2007), S. 29.
6 Vgl. Manning u.a. (2008), S. 321.
7 Vgl. Manning u.a. (2008), S. 326.

Die Zielfunktion misst dabei die Qualität des Clusterings[1] und soll optimiert werden[2], wobei sich die Optimierung in den meisten Fällen entweder auf die Ähnlichkeit oder die Distanz der Dokumente bezieht[3].
Beim Clustern handelt es sich um eine Technik aus dem Bereich des un-über-wachten Lernens (englisch: *unsupervised learning*).[4] Die Clusteringstruktur wird aufgrund des Inhalts der vorhandenen Dokumente erzeugt, ohne zusätzliche Informationen einzubeziehen.[5] Es wird versucht, die "natürliche" Unterteilung der Dokumentenmenge in Gruppen aus den Daten abzuleiten, ohne vorher durch Menschen eine Klassenstruktur zu definieren.
Bei diesem Punkt handelt es sich um das wichtigste Merkmal zur Abgrenzung des Clusterns vom Klassifizieren von Dokumenten: Innerhalb eines Klassifizierungsvorgangs wird versucht, eine zuvor festgelegte Klassenstruktur zu replizieren, d.h., Dokumente vorher festgelegten Klassen zuzuordnen. Im Bereich des Klassifizierens wird also nicht versucht, diese Gruppierung anhand der vorliegenden Dokumente zunächst zu ermitteln. Aus diesem Grund wird das Klassifizieren dem überwachten Lernen zugeordnet.[6]
Dadurch, dass beim Clustern keine vordefinierte Klassenstruktur repliziert werden soll, sondern stattdessen aufgrund der Ähnlichkeit oder Unähnlichkeit der Dokumente Cluster erzeugt werden sollen, entfällt beim Clustern die Einteilung der Dokumente in Trainings- und Testdaten. Der „Clusterer"[7] wird nicht erst trainiert und dann auf seine Qualität mit ihm zuvor unbekannten Dokumenten überprüft, sondern er erzeugt direkt mit unbekannten Dokumenten ein Clustering, welches qualitativ überprüft wird. Aus diesem Grund existieren beim Clustern nur Testdaten. Der Ablauf eines Clusterns ist in Abbildung 2.4 zu sehen.
Die Vorteile eines Clusterns von Dokumenten liegen klar auf der Hand:

- Menschen denken in Klassen, sie gruppieren unbewusst die ganze Zeit, da das ein wesentlicher Bestandteil menschlichen Denkens ist.[8]

- Die Übersichtlichkeit wird durch das Gruppieren erhöht, dadurch kann ein Mensch weitere Verfeinerungen oder Sortierungen vornehmen.

1 Vgl. Manning u.a. (2008), S. 326.
2 Vgl. Feldman u.a. (2007), S. 84, 92.
3 Vgl. Manning u.a. (2008), S. 326.
4 Vgl. Feldman u.a. (2007), S. 82.
5 Vgl. Feldman u.a. (2007), S. 82.
6 Vgl. Manning u.a. (2008), S. 321.
7 In Anlehnung an den Begriff „Klassifizierer" als Bezeichnung für die Komponente der Software, die das Klassifizieren durchführt, wird im Zusammenhang mit dem Clustern in der vorliegenden Arbeit ein analoger Begriff verwendet.
8 Vgl. Slonim u.a. (2005), S. 18297.

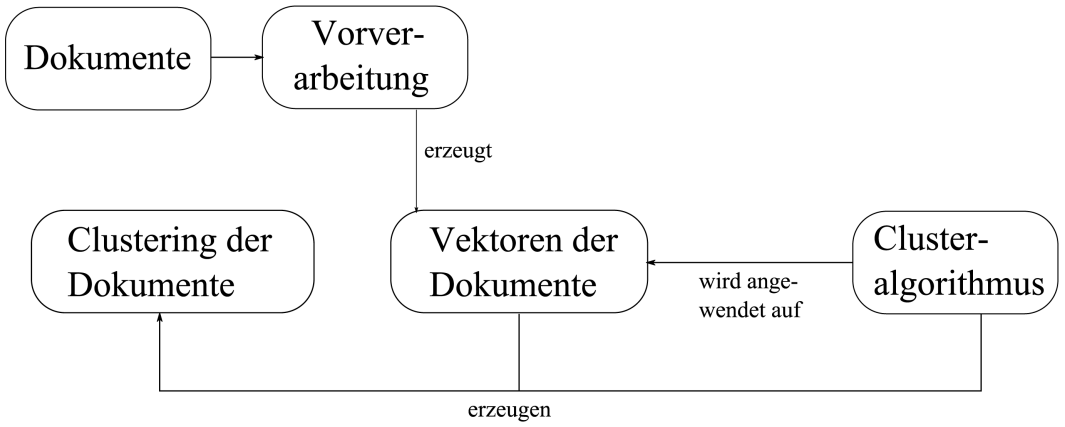


Abbildung 2.4: Ablauf eines Clusterns

- Durch die Tatsache, dass es sich um eine unüberwachte Lernmethode handelt, ist eine entstehende Gruppierung als "intersubjektiv überprüfbar" zu bezeichnen, das heißt, die Struktur ist aus den Dokumenten entstanden und nur indirekt - über die Auswahl des verwendeten Cluster-Algorithmus und seiner Parameter - von einem Menschen erzeugt, der bestimmte Schwerpunkte setzt oder eine Meinung vertritt, und der Algorithmus zur Strukturerzeugung kann nachvollzogen und die Ergebnisse können reproduziert[1] werden.

- Durch das Zusammenfassen der Dokumente zu Clustern entsteht eine Kompression dieser Dokumentenmenge[2], die dazu führt, dass bestimmte weitergehende Aktionen auf ihr leichter durchzuführen sind. Beispielsweise muss nicht jedes einzelne Element gelöscht werden, sondern es kann die gesamte Gruppe gelöscht werden, wenn die Dokumente nicht relevant sind.

2.4.2 Ansätze zum Clustern

Erstellt man ein Clustering, so kann dies auf verschiedene Arten erfolgen. Für einen Überblick über die Ansätze wird auf eine hierarchische Darstellung nach Jain u.a.[3] verwiesen. Diese befindet sich in Abbildung 2.5 auf S. 63.

1 Das gilt uneingeschränkt jedoch nur, wenn der Algorithmus deterministisch ist.
2 Vgl. Slonim u.a. (2005), S. 18297.
3 Vgl. Jain u.a. (1999), S. 275. Die Grafik wurde übersetzt und es wird durch die Pünktchen angedeutet, dass in der Zeit seit der Entstehung noch weitere Ansätze hinzugekommen sind. Darin eingeschlossen ist der ergänzte GAAC-Algorithmus, der hier namentlich genannt wird, weil er einer der implementierten Algorithmen ist (siehe Kapitel 2.4.3.2).

Der Hauptunterschied bei den Ansätzen liegt in der sich ergebenden Struktur des Ergebnisses. So entstehen beim *flachen* oder *partitionierenden* Clustern Cluster, die untereinander keine Beziehung haben. Es werden also Cluster aus den Dokumenten gebildet, indem ähnliche Dokumente zusammengefasst werden, aber keine Ähnlichkeiten zwischen den Clustern berücksichtigt werden.

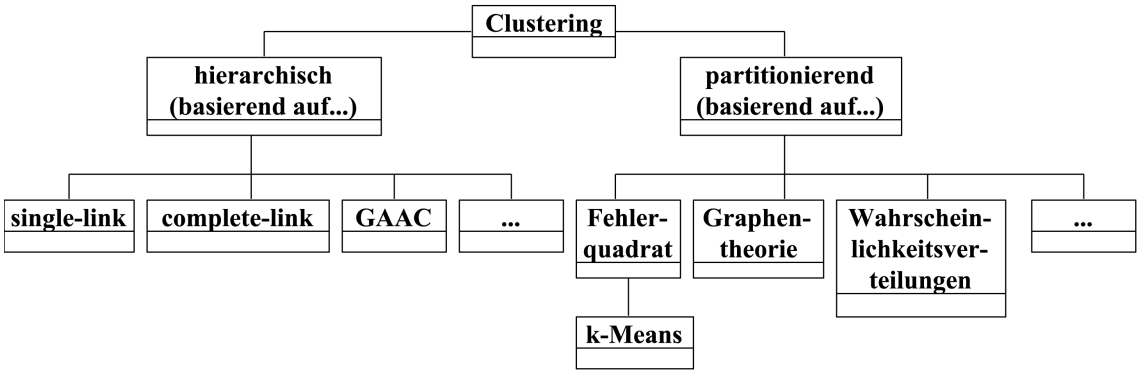


Abbildung 2.5: Hierarchie der Ansätze zum Clustern

Beim *hierarchischen* Clustern dagegen soll eine Struktur entstehen, die die Cluster in eine hierarchische Beziehung zueinander setzt, also darauf basiert, dass auch Ähnlichkeiten zwischen Clustern berücksichtigt werden.[1]
Die Hierarchie allein ist jedoch nicht ausreichend, um alle Ansätze vollständig darstellen zu können. Neben ihr existieren weitere Unterteilungsmöglichkeiten, die jedoch nicht ohne Weiteres in sie einfließen können, da sie „quer“ zu den in der Hierarchie aufgeführten Ansätzen liegen. Dazu zählen die im Weiteren aufgeführten Merkmale.[2] Soll ein hierarchisch aufgebautes Clustering erzeugt werden, existieren zwei Möglichkeiten bei der Vorgehensweise:

- agglomerierend (*agglomerative clustering*)
  Es werden Cluster über das Zusammenfassen von Dokumenten gebildet. Zu Anfang bildet jedes Dokument einen Cluster. Nach und nach werden ähnliche Dokumente zu Clustern zusammengefasst und ähnliche Cluster zu neuen Clustern vereinigt. Es ist möglich, unterschiedliche Abbruchkriterien zu definieren: Im einfachsten Fall ist die Verarbeitung beendet, wenn nur noch ein Cluster vorhanden ist. Des Weiteren kann aber auch eine bestimmte Clusteranzahl

1 Auf die weiteren Ebenen der abgebildeten Hierarchie wird hier nicht eingegangen. Sie wurden der Vollständigkeit halber und zur Verdeutlichung, wo die verwendeten Algorithmen, siehe Kapitel 2.4.3, in der Hierarchie stehen, mit aufgeführt.
2 Für eine vollständige Darstellung vgl. Jain u.a. (1999), S. 274 f.

vorgegeben werden oder aber ein Qualitätskriterium. Dieses unterbricht die Verarbeitung, wenn ein bestimmter Ähnlichkeitslevel nicht mehr überschritten wird oder aber der Abstand zwischen den noch vorhandenen Clustern zu groß wird, sie sich also zu unähnlich werden, um sie zusammenzulegen. Diese Art des Clusterns bezeichnet man auch als *bottom-up* oder *hierarchical agglomerative clustering*.

- unterteilend (*divisive clustering*)
  In diesem Fall verhält es sich genau umgekehrt. Alle Dokumente in der Ausgangslage werden als ein großer Cluster betrachtet, den es zu zerteilen gilt. Dafür muss nicht mehr die Ähnlichkeit zwischen Dokumenten oder Clustern gemessen werden, sondern die Unähnlichkeit. In jedem Schritt erfolgt hier so lange eine Unterteilung in Cluster, bis jedes Dokument einen eigenen Cluster bildet. Auch hier können andere Abbruchkriterien verwendet werden. Diese Art des Clusterns bezeichnet man als *top-down clustering*.

In dieser Arbeit werden beide Varianten - also partitionierendes und hierarchisches Clustering - in den Experimenten verwendet.[1]
Ein weiteres Unterscheidungsmerkmal in Bezug auf die Art des Clusterings ist das Differenzieren in hart und weich. Von einem *harten* Clustering[2] spricht man, wenn jedes Dokument genau einem Cluster zugeordnet ist.[3] Ein *weiches* Clustering erlaubt dagegen die Zuordnung eines Dokuments zu mehreren Clustern[4], da bspw. auf inhaltlicher Ebene mehrere Themen in diesem Dokument behandelt werden.
Die formale Definition eines harten Clusterings ist:

$$\omega_i \cap \omega_j = \emptyset \text{ für alle } i, j \in \{1, ..., c\}$$

Für den Fall eines weichen Clusterings gilt:

$$\omega_i \cap \omega_j \neq \emptyset \text{ für mindestens ein } i \in \{1, ..., c\} \text{ und}$$
$$\text{mindestens ein } j \in \{1, ..., c\} \text{ mit } i \neq j$$

In den Experimenten der vorliegenden Arbeit werden nur harte Clusterings erzeugt.[5]

1 Siehe Kapitel 5.3 dieser Arbeit.
2 Vgl. Manning u.a. (2008), S. 322.
3 Hierbei muss man jedoch berücksichtigen, dass diese Aussage für das hierarchische Clustering so nicht gelten kann, da jedes Dokument per definitionem auch zu allen übergeordneten Clustern gehört. Hier müsste die Aussage so umformuliert werden, dass gilt: Jedes Dokument wird genau einem Cluster pro Ebene zugeordnet.
4 Vgl. Manning u.a. (2008), S. 322.
5 Siehe Kapitel 5.3.

2.4.3 Algorithmen für das Clustern

2.4.3.1 k-Means-Algorithmus

Ein Algorithmus, um ein partitionierendes Clustering zu erzeugen, ist der k-Means-Algorithmus[1]. Im Idealfall erzeugt er Cluster mit einem Clustermittelpunkt als Schwerpunkt und den Dokumenten mit der geringsten Distanz zu diesem Clustermittelpunkt.
Der Ablauf des Algorithmus ist der Folgende[2]:

1. Wähle zufällig k[3], in der vorliegenden Arbeit c, Feature-Vektoren der Dokumente d_i aus der Menge der Dokumente D aus. Diese Vektoren bilden die ersten Clustermittelpunkte.

2. Wiederhole, bis das Abbruchkriterium erfüllt ist:

   a) Ordne jedes Dokument d_i dem Cluster ω_j zu, zu dessen Clustermittelpunktvektor $\vec{\mu}$ der Vektor $\vec{f}_{d_i}$ die geringste Distanz aufweist - also dem Clustermittelpunktvektor, dem sie am ähnlichsten sind.[4]

   b) Berechne die Clustermittelpunktvektoren neu, indem der Durchschnitt über alle im Cluster vorhandenen Feature-Vektoren gebildet wird.
   Als mathematische Formel geschrieben, ergibt sich Folgendes:
   $\vec{\mu}(\omega) = \frac{1}{|\omega|} \sum_{\vec{f}_{d_i} \in \omega} \vec{f}_{d_i}$.

1 Die Ursprünge dieses Algorithmus reichen bis in die 50er Jahre zurück und lassen sich laut Bock (2008), S. 2, auf mehrere Wissenschaftler zurückführen, die ihn fast zeitgleich sehr ähnlich entwickelten. Zwei wichtige Entwicklungen werden hier in den Vordergrund gestellt: Die erste Erwähnung des Namens *k-means algorithm* von MacQueen (vgl. Bock (2008), S. 6, und MacQueen (1967), S. 281) und eine andere Benennung in der Informatik: *Lloyd's algorithm I*(vgl. Bock (2008), S. 7, und Lloyd (1982), S. 131 f.), wobei dieser Algorithmus bereits aus dem Jahr 1957 stammt, vgl. Lloyd (1982), S. 136.

2 Die Originalbeschreibung des Ablaufs befindet sich in MacQueen (1967), S. 283. Sie wurde an das hier beschriebene Problem angepasst, indem anstatt Datenpunkte Dokumentenvektoren verwendet werden und anstatt Gruppen Cluster.

3 In der vorliegenden Arbeit ist der Bezeichner k bereits im Zusammenhang mit dem Klassifizieren von Dokumenten beim k-Nearest-Neighbour-Algorithmus verwendet worden. Dort bezeichnet k die Anzahl der Nachbarn, aufgrund derer ein Dokument einer Klasse zugeordnet wird. Im Zusammenhang mit dem k-Means-Algorithmus im Clustering bezeichnet k die Anzahl der entstehenden Cluster, die bisher mit c bezeichnet wurde. Da es sich jedoch um feststehende Namen für die Algorithmen handelt, muss an dieser Stelle eine Doppeldeutigkeit bei den Bezeichnungen in Kauf genommen werden.

4 Es kann der Fall auftreten, dass der Dokumentenvektor zu *zwei oder mehr* Clustermittelpunktvektoren die geringste Distanz aufweist. In diesem Fall wird hier wie bei Arthur u.a. (2005), S. 2, davon ausgegangen, dass eine zufällige Zuordnung zu einem dieser Clustermittelpunkte stattfindet. Durch die Neuberechnung der Clustermittelpunktvektoren ist es unwahrscheinlich, dass dieser Fall in der nächsten Iteration für den gleichen Feature-Vektor eines Dokuments erneut auftritt.

Das bedeutet, der Vektor $\vec{\mu}$ des Clustermittelpunkts des Clusters ω ergibt sich aus der Summe über alle im Cluster vorhandenen Vektoren $\vec{f}_{d_i}$ geteilt durch die Anzahl der im Cluster vorhandenen Dokumente.

Als Algorithmus zusammengefasst ergibt sich:

Algorithmus 4 k-Means-Algorithmus

Eingabe: Menge der Dokumente D, Anzahl der Cluster k (hier c), Abbruchkriterium sk, Distanzmaß dm

Ausgabe: Clusterzuordnung für jedes Dokument aus D

```
1: Wähle zufällig c Dokumente aus D aus, das ist die Menge CZ
2: Erzeuge c Cluster mit jeweils einem Element aus CZ als Inhalt, das ist Ω
3: while sk ≠ true do
4:     for all Dokument d_i ∈ D do
5:         minDist = ∞
6:         for all Clustermittelpunkt cz ∈ CZ do
7:             dist = dm(d_i, cz)
8:             if dist ≤ minDist then
9:                 if dist < minDist then
10:                    minDist = dist
11:                    Merke aktuelles Cluster als zuzuordnendes, das ist cluster
12:                else
13:                    Hänge aktuelles Cluster an Liste von zuzuordnenden Clustern
                       an
14:                end if
15:            end if
16:        end for
17:        if mehr als ein zuzuordnendes Cluster then
18:            Wähle randomisiert ein Cluster aus und weise es cluster zu
19:        end if
20:        Ordne das Dokument d_i dem Cluster cluster zu
21:    end for
22:    Leere CZ
23:    for all Cluster cluster ∈ C do
24:        Berechne den neuen Clustermittelpunkt cz
25:        Weise cz CZ zu
26:    end for
27: end while
```

Zur Berechnung der Distanz wird zumeist das euklidische Distanzmaß verwendet. Es existieren aber auch andere Distanzmaße.[1]
Für das erwähnte Abbruchkriterium liegen verschiedene Möglichkeiten vor, die jeweils Vor-, aber auch Nachteile haben[2]:

- Es wird eine feste Anzahl von Iterationen angegeben.
  Vorteil: Die Laufzeit des Algorithmus ist vorgegeben.
  Nachteil: Die Ergebnisqualität[3] kann gering sein.

- Die Dokumentzuordnung zu den Clustern in zwei unmittelbar aufeinanderfolgenden Iterationen bleibt unverändert.
  Vorteil: Man erhält ein gutes[4] Clusteringergebnis.
  Nachteil: Das Auftreten sehr langer Laufzeiten ist möglich.[5]

- Die Clustermittelpunktvektoren bleiben in zwei unmittelbar aufeinanderfolgenden Iterationen unverändert.
  Vorteil: Man erhält ein gutes[6] Clusteringergebnis.
  Nachteil: Das Auftreten sehr langer Laufzeiten ist möglich.[7]

Zusammenfassend kann gesagt werden, dass bei allen drei Möglichkeiten abgewogen werden muss, ob mehr Wert auf kurze Laufzeiten oder ein gutes Clustering gelegt wird.
Die Auswahl des Abbruchkriteriums ist jedoch nicht die einzige Schwierigkeit bei der Implementierung des Algorithmus. Bereits für den ersten Schritt - die Auswahl der c Clustermittelpunktvektoren - ergeben sich zwei Probleme:

1. Die Anzahl c der Cluster muss vorher festgelegt werden. Ohne Kenntnis der zu clusternden Dokumente muss diese Anzahl „geraten“ werden.

2. Die ersten Clustermittelpunktvektoren werden zufällig ausgewählt. Hier kann eine „schlechte“ Auswahl[8] getroffen werden.

1 Für einen Überblick über Ähnlichkeits- und Distanzmaße siehe Kapitel 2.2 dieser Arbeit. Das jeweils verwendete Distanzmaß wird im Kapitel 5.3.3 mit den Experimentbeschreibungen aufgelistet.
2 Vgl. Manning u.a. (2008), S. 332 f.
3 Eine Spezifikation von Qualitätskriterien wird in Kapitel 2.5.3, S. 78, aufgestellt.
4 Siehe Fußnote 3.
5 Endlosschleifen können hier nicht auftreten, da der k-Means-Algorithmus in jedem Fall zu einem lokalen Minimum konvergiert und somit terminiert, vgl. Manning u.a. (2008), S. 334, und Arthur u.a. (2005), S. 1.
6 Siehe Fußnote 3.
7 Siehe Fußnote 5.
8 Damit ist eine Auswahl gemeint, die ein Clustering erzeugt, das als „schlecht“ gilt. Zur Evaluation eines erzeugten Clusterings siehe Kapitel 2.5.3.

Eine vorher festzulegende Anzahl an Clustern ist ein Nachteil des k-Means-Algorithmus, wie auch anderer Algorithmen zum Erstellen flacher Clusterings, da sie ohne vorherige Kenntnis der Dokumente nicht plausibel "geraten" werden kann.[1] Es existieren Ansätze, die menschliches Vorwissen über die Daten nutzen[2], und andere, die durch Heuristiken versuchen, die Anzahl plausibel festzulegen[3], jedoch wird hier nicht näher darauf eingegangen, da in den Experimenten, siehe Kapitel 5.3.3 dieser Arbeit, eine Algorithmus-Implementierung verwendet wird, die eine vorgegebene Anzahl an Clustern erwartet.
Der große Vorteil des k-Means-Algorithmus ist seine einfache Implementierbarkeit und seine garantierte Konvergenz.[4] Das ist aber gleichzeitig mit einem Nachteil verbunden, da nicht garantiert werden kann, dass das globale Optimum erreicht wird.[5] Eine mögliche Begründung für das Nicht-Erreichen des globalen Optimums ist die Auswahl der "falschen" Clustermittelpunkte im ersten Schritt. "Falsch" heißt hier, dass zum Beispiel ein so genannter "Ausreißer" als Mittelpunkt gewählt wird. Dadurch können Cluster mit nur einem Dokument entstehen, da "Ausreißer" Dokumente darstellen, die wenig Ähnlichkeit mit allen anderen Dokumenten aufweisen, so dass es also auch sehr unwahrscheinlich ist, dass diesem Cluster weitere Dokumente zugewiesen werden. Einige Heuristiken, um solche Situationen zu vermeiden, sind[6]:

- Ausschließen von Ausreißern aus der Menge von Dokumenten, aus der die ersten Clustermittelpunkte gewählt werden,
- Erzeugen mehrerer Startpositionen und Auswahl der besten Startposition sowie
- Ableiten der Menge an möglichen ersten Clustermittelpunkte aus anderen Clusteralgorithmen, zum Beispiel aus dem hierarchischen Clustern.

2.4.3.2 Group-average-agglomerative-clustering-Algorithmus

Der Group-average-agglomerative-clustering-Algorithmus (GAAC), übersetzt etwa: zusammenfassendes Gruppendurchschnitts-Clustern, gehört zur Gruppe der hierar-

1 Vgl. Manning u.a. (2008), S. 327.
2 Dazu zählen beispielsweise das Festlegen der Anzahl durch Experten oder Benutzer.
3 Dazu zählt das Testen verschiedener Werte für die Anzahl und Auswahl der Anzahl, die bezogen auf ein Qualitätskriterium, das vielversprechendste Ergebnis liefert, vgl. bspw. Pelleg u.a. (2000), S. 727-734; Manning u.a. (2008), S. 336-338; Liu u.a. (2002), S. 191-198.
4 Vgl. Manning u.a. (2008), S. 334.
5 Vgl. Manning u.a. (2008), S. 334.
6 Vgl. Manning u.a. (2008), S. 335.

chischen, agglomerierenden Clusteralgorithmen.[1] Wie bereits im Namen des Algorithmus enthalten, verwendet er für die Entscheidung, welche Cluster zusammengefasst werden sollen, den Gruppendurchschnitt.

Um Cluster bilden zu können und auch später zwei Cluster zu einem zusammenfassen zu können, wird bei diesem Algorithmus nicht die paarweise Ähnlichkeit zwischen zwei beispielhaft ausgewählten Dokumentvektoren aus verschiedenen Clustern betrachtet, wie es beim *single-link clustering* oder beim *complete-link clustering*[2] der Fall ist, sondern es werden die paarweisen Ähnlichkeiten *aller* Cluster berücksich-

1 Vgl. Lance u.a. (1967), S. 375; Willett (1988), S. 582; Voorhees (1986), S. 466.

2 Beim single-link clustering wird die Ähnlichkeit zweier Cluster über ihre ähnlichsten Mitglieder bestimmt. Das bedeutet, es werden jeweils nur zwei Dokumente pro Clusterpaar betrachtet, und zwar die, die sich am ähnlichsten sind, um die Ähnlichkeit der beiden Cluster zu bestimmen. Das gleiche gilt für das complete-link clustering, jedoch werden hier die beiden Dokumente betrachtet, die sich am unähnlichsten sind. Also werden die Cluster vereinigt, deren „Durchmesser“, wenn man die beiden unähnlichsten Dokumente, betrachtet über alle Clusterpaare, grafisch durch eine Linie miteinander verbinden würde, am kleinsten ist, vgl. Manning u.a. (2008), S. 350-353; Voorhees (1986), S. 466. In der nachfolgenden Abbildung, entnommen Manning u.a. (2008), S. 350, werden die vier Möglichkeiten, die Ähnlichkeit zwischen Clustern beim hierarchischen Clustern zu bestimmen, dargestellt. Die noch nicht erwähnte Möglichkeit der Ähnlichkeitsbestimmung über die Clustermittelpunkte ist in Abbildungsteil (c) zu sehen. Die Ähnlichkeiten werden paarweise über alle Clustermittelpunkte bestimmt und die ähnlichsten Cluster zusammengefasst.

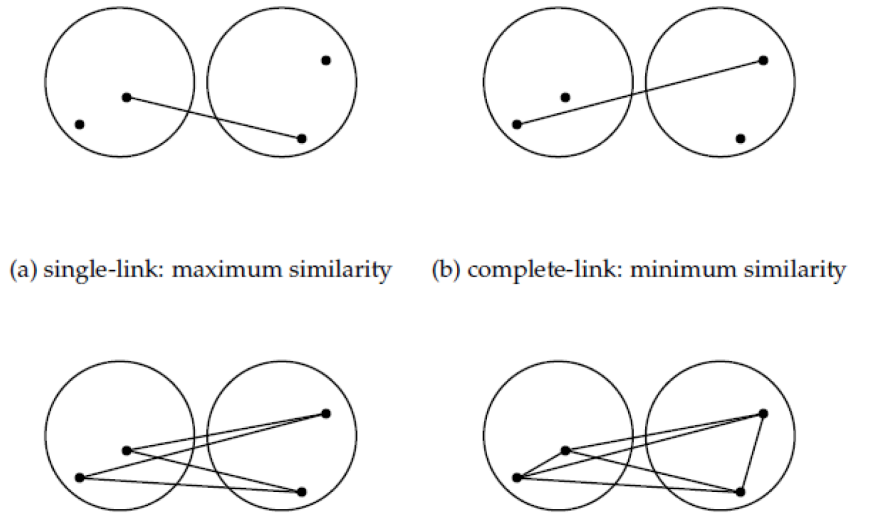


Abbildung 2.6: Möglichkeiten der Ähnlichkeitsbestimmung beim hierarchischen Clustern

tigt. Es wird also keine Reduzierung auf die beiden ähnlichsten oder die beiden unähnlichsten Cluster vorgenommen, wodurch einige Probleme, die in den beiden genannten Verfahren auftreten können[1], vermieden werden. Berechnet wird diese Ähnlichkeit wie folgt[2]:

$$\text{group-average similarity}(\omega_a, \omega_b) = \frac{1}{(|\omega_a|+|\omega_b|)(|\omega_a|+|\omega_b|-1)} \sum_{\vec{f}_{d_i} \in \omega_a \cup \omega_b} \sum_{\vec{f}_{d_{i'}} \in \omega_a \cup \omega_b, d_{i'} \neq d_i} \vec{f}_{d_i} \cdot \vec{f}_{d_{i'}}$$

Die Ähnlichkeit zwischen zwei Clustern ω_a und ω_b wird folglich berechnet, indem die Summe der Skalarprodukte aller vorhandenen Vektoren in den beiden Clustern - ohne die Ähnlichkeit zu sich selbst zu berücksichtigen - durch die Anzahl der vorhandenen Vektoren (fast[3]) zum Quadrat geteilt wird.[4]
Bei diesem Algorithmus wird der große Nachteil der flachen Clusteralgorithmen aufgehoben: eine vorher definierte Anzahl an Clustern ist unnötig, jedoch kann eine Anzahl als Abbruchkriterium definiert werden.
Wie bereits in Kapitel 2.4.2 beschrieben, läuft der Algorithmus so ab, dass aus vielen Clustern - jedes Dokument befindet sich am Anfang in einem eigenen Cluster - Schritt für Schritt weniger werden, bis das Abbruchkriterium erreicht ist. Mögliche Abbruchkriterien sind:

- Wird ein vorher festgelegter Ähnlichkeitslevel zwischen den noch vorhandenen Clustern und Dokumenten erreicht, so wird die Verarbeitung abgebrochen und das weitere Zusammenfassen beendet.
- Die vorgegebene Anzahl der Cluster wird erreicht.

Der Algorithmus läuft nach folgenden Schritten ab, wobei jedes Dokument zunächst jeweils einen eigenen Cluster bildet[5]:

Wiederhole, bis das Abbruchkriterium erreicht ist:

1. Berechne die paarweisen Ähnlichkeiten zwischen allen vorhandenen Clustern.

1 Vgl. Jain u.a. (1999), S. 276; Manning u.a. (2008), S. 352 f.
2 Vgl. Manning u.a. (2008), S. 356.
3 Das “fast” entsteht, weil die Ähnlichkeit zu sich selbst nicht berücksichtigt wird.
4 Wie bereits im Kapitel 2.2.2 erläutert, kann die Ähnlichkeit zweier Vektoren über den Cosinus zwischen ihnen bestimmt werden. Zur Berechnung des Cosinus wird die Formel 2.2 verwendet. Im Fall der group-average similarity werden normalisierte Vektoren verwendet, so dass die Cosinus-Formel auf das Skalarprodukt der Vektoren reduziert werden kann, vgl. Manning u.a. (2008), S. 111.
5 Vgl. Voorhees (1986), S. 470.

2. Fasse die beiden ähnlichsten Cluster zu einem gemeinsamen Cluster zusammen.[1]

Bilde optional das Clusteringergebnis in einem Dendrogramm[2] ab.

Eine formale Darstellung des Algorithmus befindet sich in Manning u.a. (2008), S. 354, und Voorhees (1986), S. 471.

2.5 Theoretische Grundlagen der Evaluation von Klassifizierungs- und Clusterergebnissen

2.5.1 Einführung in die Evaluation von Klassifizierungs- und Clusterergebnissen

In der vorliegenden Arbeit wird das automatische Klassifizieren und Clustern von natürlichsprachlichen Texten mit Hilfe einer neuartigen Repräsentation der Texte betrachtet. Um die Qualität der neuartigen Aufbereitung zu messen, werden die erreichten Ergebnisse evaluiert. Das geschieht im Vergleich zu einer wortbasierten Aufbereitung der Texte.
Die verwendeten Algorithmen weisen den Testdokumenten Klassen bei der Klassifizierung oder Cluster beim Clustern zu. Um die Qualität dieser Zuweisung, und damit indirekt auch der verwendeten Aufbereitung, prüfen zu können, werden verschiedene Maße in der vorliegenden Arbeit verwendet. Diese stellen die zugewiesenen Klassen oder Cluster den tatsächlichen Klassen oder Clustern, denen die Dokumente angehören, gegenüber. Durch die Berechnung eines Evaluationswertes kann dann die Qualität festgestellt werden.
Bevor auf die einzelnen Maße eingegangen wird, werden hier einige Begriffe erläutert.

Goldstandards

Um eine erzeugte Klassifikation oder ein Clustering zu evaluieren, verwendet man so genannte *Goldstandards*. Das sind Dokumentsammlungen, für die von Experten bereits Klassen vordefiniert wurden. Die Dokumente der Dokumentsammlungen

1 In der verwendeten Implmentierung werden jeweils die ersten beiden Cluster, bei denen die größte Ähnlichkeit gemessen wurde, zu einem neuen Cluster zusammengefasst. Weitere Cluster mit der gleichen größten Ähnlichkeit werden zunächst nicht vereinigt. Im Algorithmus in Voorhees (1986), S. 471, wird das ebenfalls so gehandhabt.

2 Zur besseren Nachvollziehbarkeit des Ablaufs des Clusterns bietet es sich bei hierarchischen Algorithmen an, das Ergebnis als so genanntes Dendrogramm darzustellen. Ein Dendrogramm visualisiert das Ergebnis eines erfolgten hierarchischen Clusterns - siehe Abbildung 2.10 auf S. 86.

wurden aufgrund dieser vordefinierten Klassen klassifiziert[1] und können geclustert werden, um dann das Ergebnis mit den vordefinierten Klassen zu vergleichen. Eine Auswahl dieser Standard-Dokumentensammlungen findet man in Manning u.a. (2008).[2]
Die Qualität der erzeugten Klassifikation oder des erzeugten Clusterings wird mit Hilfe der Dokumentsammlung evaluiert. Hierzu werden so genannte externe Kriterien verwendet, die sich im Fall einer Klassifizierung auf die Qualität der erzeugten Klassifikation relativ zur vorgegebenen Klassifikation und im Fall eines Clusterns auf die Qualität des erzeugten Clusterings relativ zur vorgegebenen Klassifikation beziehen. Sie messen hier, inwieweit die erzeugte Klassifikation bzw. das erzeugte Clustering - also die Menge der erzeugten Cluster - den als „korrekte“ Klassifikation oder als „korrektes“ Clustering zu Grunde gelegten Goldstandard abbildet.

Bestandteile der Qualitäsmaße
Fasst man ein Klassifizieren oder ein Clustern als Abfolge von nacheinander getroffenen Entscheidungen auf, die Dokumente einer Klasse, im Fall einer Klassifizierung, oder die Dokumente Clustern, im Fall eines Clusterns, zuzuordnen, so existieren jeweils vier Möglichkeiten, die Zuordnung in Bezug auf eine gerade betrachtete vordefinierte Klasse zu interpretieren:
Klassifizieren

(a) Ein Dokument wird seiner in der Goldstandard-Klassifikation zugeordneten Klasse beim Klassifizieren zugeordnet. Dabei handelt es sich um eine richtige Entscheidung. Die Bezeichnung für ein Dokument, auf welches der Sachverhalt zutrifft, ist *true positive*.[3]

(b) Ein Dokument wird einer in der Goldstandard-Klassifikation nicht zugeordneten Klasse beim Klassifizieren nicht zugeordnet. Auch in diesem Fall handelt es sich um eine richtige Entscheidung. Ein Dokument, auf welches der Sachverhalt zutrifft, wird als *true negative* bezeichnet.[4]

(c) Ein Dokument wird einer in der Goldstandard-Klassifikation nicht zugeordneten Klasse beim Klassifizieren trotzdem zugeordnet. In diesem Fall handelt es

1 Sie werden deshalb hauptsächlich beim Klassifizieren von Texten eingesetzt.
2 Vgl. Manning u.a. (2008), S. 141 f.
3 Im weiteren Verlauf der Arbeit bezeichnet TP die Anzahl der Dokumente, die true positive sind.
4 Im weiteren Verlauf der Arbeit bezeichnet TN die Anzahl der Dokumente, die true negative sind.

sich um eine falsche Entscheidung. Ein Dokument, auf das dieser Sachverhalt zutrifft, wird als *false positive* bezeichnet.[1]

(d) Ein Dokument wird einer in der Goldstandard-Klassifikation zugeordneten Klasse beim Klassifizieren nicht zugeordnet. Hier wird eine falsche Entscheidung getroffen. Die Bezeichnung für ein Dokument, auf welches der Sachverhalt zutrifft, ist *false negative*.[2]

Clustern

(a) Zwei einander ähnliche Dokumente werden dem gleichen Cluster zugeordnet. Dabei handelt es sich um eine richtige Entscheidung. Die Bezeichnung für Dokumente, die diesen Sachverhalt erfüllen, ist *true positive*.

(b) Zwei nicht-ähnliche Dokumente werden unterschiedlichen Clustern zugeordnet. Auch in diesem Fall handelt es sich um eine richtige Entscheidung. Die Bezeichnung für Dokumente, die diesen Sachverhalt erfüllen, ist *true negative*.

(c) Zwei unähnliche Dokumente werden dem gleichen Cluster zugeordnet. In diesem Fall handelt es sich um eine falsche Entscheidung.Die Bezeichnung für Dokumente, die diesen Sachverhalt erfüllen, ist *false positive*.

(d) Zwei ähnliche Dokumente werden unterschiedlichen Clustern zugeordnet. Hier wird eine falsche Entscheidung getroffen. Die Bezeichnung für Dokumente, die diesen Sachverhalt erfüllen, ist *false negative*.

Um diese Anzahlen berechnen zu können, sind folgende Festlegungen nötig[3]:

- Es existiert eine Menge L vorher definierter Klassen. Die Anzahl der in der Menge enthaltenen Klassen wird mit $|L|$ bezeichnet.
- Jedes der zu klassifizierenden oder zu clusternden Dokumente gehört zu genau einer vordefinierten Klasse L_i.
- Jedes der zu klassifizierenden oder zu clusternden Dokumente wird genau einer Klasse[4] bzw. einem Cluster[5] ω_j zugeordnet.

1 Im weiteren Verlauf der Arbeit bezeichnet FP die Anzahl der Dokumente, die false positive sind.

2 Im weiteren Verlauf der Arbeit bezeichnet FN die Anzahl der Dokumente, die false negative sind.

3 Vgl. Forster (2006), S. 45 f. Um eine einheitliche Schreibweise beizubehalten, wurden Bezeichner angepasst.

4 In der vorliegenden Arbeit wird eine Single-label-multi-class-Klassifizierung durchgeführt.

5 Im Fall eines harten Clusterings gehören die Dokumente zu *genau* einem Cluster. Betrachtet man den Fall eines weichen Clusterings, gehört ein Dokument zu *mindestens* einem Cluster.

- $h(L_i, \omega_j)$ stellt die Anzahl der Dokumente in der zugewiesenen Klasse oder dem zugewiesenen Cluster ω_j dar, die zur vordefinierten Klasse L_i gehören.

Im Fall einer Klassifizierung werden die oben angegebenen Sachverhalte (a) bis (d), aus denen sich die jeweiligen Anzahlen TP, TN, FP und FN ergeben, durch das Zählen der Übereinstimmungen oder Nicht-Übereinstimmungen der erzeugten Klassifikation mit der Goldstandard-Klassifikation ermittelt.
Für ein Clustering werden die oben angegebenen Sachverhalte (a) bis (d),aus denen sich die jeweiligen Anzahlen TP, TN, FP und FN ergeben, nach Dom (2001) und Forster (2006)[1] wie folgt berechnet[2]:

(a) $TP = \sum_{i=1}^{|L|} \sum_{j=1}^{c} \binom{h(L_i,\omega_j)}{2}$

(b) $TN = \binom{|D|}{2} - (TP + FN + FP)$

(c) $FN = \sum_{i=1}^{|L|} \binom{|L_i|}{2} - TP$

(d) $FP = \sum_{j=1}^{c} \binom{|\omega_j|}{2} - TP$

2.5.2 Evaluation der Klassifizierungsergebnisse

2.5.2.1 Anteil der korrekt klassifizierten Dokumente

Bei diesem Maß wird das Verhältnis gebildet zwischen der Anzahl der korrekt klassifizierten Dokumente und der Gesamtanzahl der zu klassifizierenden Dokumente. Ein korrekt klassifiziertes Dokument ist ein Dokument, dem eine Klasse durch einen Klassifizierungsalgorithmus zugewiesen wurde, die es auch im Goldstandard hat. Es ergibt sich also[3]:

$$\text{korrekter Anteil} = \frac{TP}{|D|}$$

2.5.2.2 F-Maß für die Klassifizierungsergebnisse

Das F-Maß[4] ermöglicht es, die Qualität der Ergebnisse differenzierter zu betrachten als der korrekte Anteil. Dazu kann eine Gewichtung angegeben werden, ob eher auf die Genauigkeit oder aber auf die Trefferquote Wert gelegt wird.

1 Vgl. Dom (2001), S. 3 f., und Forster (2006), S. 46.
2 Dabei ist $\binom{x}{2}$ der Binomialkoeffizient „2 aus x“. Für den Binomialkoeffizienten gilt, vgl. Bronstein u.a. (2005), 13 f.: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, wobei $n, k \in \mathbb{Z}$ und $0 \leq k \leq n$.
3 Anmerkung: Auch die true negatives sind korrekt klassifizierte Dokumente. Will man diese einbeziehen, so ergibt sich die so genannte *accuracy*, vgl. Manning u.a. (2008), S. 143.
4 Vgl. Jardine u.a. (1971), S. 221.

Die Genauigkeit ist der Anteil der korrekt klassifizierten Dokumente, die sich in einer Klasse befinden, an der gesamten Dokumentenanzahl der jeweiligen Klasse. Bezieht man diese Aussage nicht nur auf eine Klasse, sondern auf alle Klassen, also die gesamte erzeugte Klassifikation, so versteht man darunter den Anteil der TP an der Summe aus TP und FP. Der Wert, der sich hinter den TP verbirgt, beinhaltet, wie zuvor gesagt, die Anzahl der „richtigen“ Zuordnungen von Dokumenten zu ihren Klassen des Goldstandards über alle Klassen. In den Klassen können jedoch auch Dokumente enthalten sein, die nicht in diese Klassen gehören. Die Anzahl dieser Dokumente sind die FP. Die Summe aus beiden bildet bezogen auf die gesamte Klassifikation die Summe der „richtig“ zugeordneten Dokumente und der „falsch“ zugeordneten Dokumente über alle Klassen. Die Genauigkeit für die gesamte Klassifikation ist also wie folgt definiert: $Prec = TP/(TP + FP)$.[1]

Die Trefferquote misst den Anteil der Dokumente, die sich in der gleichen Klasse befinden, an der Anzahl von Dokumenten, die sich eigentlich in der gleichen Klasse befinden sollten. Bezogen auf die gesamte Klassifikation bedeutet das den Anteil der TP an der Summe aus TP und FN. Der Wert, der sich hinter den TP verbirgt, beinhaltet, wie zuvor gesagt, die Anzahl der „richtigen“ Zuordnungen von Dokumenten zu ihren Klassen des Goldstandards über alle Klassen. In den Klassen können jedoch auch Dokumente fehlen, die eigentlich in diese Klasse gehören. Die Anzahl dieser Dokumente sind die FN. Die Summe aus beiden bildet bezogen auf die gesamte Klassifikation die Summe der „richtig“ zugeordneten Dokumente und der „fälschlicherweise“ anderen Klassen zugeordneten Dokumente über alle Klassen. Die Trefferquote für die gesamte Klassifikation ist also wie folgt definiert: $Rec = TP/(TP + FN)$.[2]

Das F-Maß bildet den harmonischen[3], gewichteten Mittelwert von Genauigkeit und Trefferquote ab. Der Gewichtungsfaktor β gibt dabei an, in welche „Richtung“ sich die Gewichtung verschiebt: eher in Richtung einer stärkeren Gewichtung der Genauigkeit oder aber in Richtung einer stärkeren Gewichtung der Trefferquote.
Die Formel zur Berechnung des F-Maßes lautet wie folgt[4]:

$$F_\beta = \frac{(\beta^2 + 1) Prec\ Rec}{\beta^2 Prec\ +\ Rec} \text{ mit } Prec\ = \frac{TP}{TP + FP} \text{ und } Rec\ = \frac{TP}{TP + FN} \tag{2.11}$$

1 Das $Prec$ steht hier für die Genauigkeit, abgeleitet aus dem englischen Wort *precision*.
2 Das Rec steht hier für die Trefferquote, abgeleitet aus dem englischen Wort *recall*.
3 Das harmonische Mittel der Zahlen $x_1, ..., x_N$ wird wie folgt berechnet: $\overline{x}_{\text{harmonisch}} = \frac{N}{\sum_{i=1}^{N} \frac{1}{x_i}}$, vgl. Bronstein u.a. (2005), S. 20. Für den hier gegebenen Fall des harmonischen Mittels von Genauigkeit und Trefferquote würde das bedeuten: $\overline{x}_{\text{harmonisch}} = \frac{2}{\frac{1}{Prec} + \frac{1}{Rec}}$.
4 Vgl. Jardine u.a. (1971), S. 221. Eine Herleitung der verwendeten Formel ausgehend vom harmonischen Mittel befindet sich im Anhang B auf S. 574. In der Originalquelle sind die Bezeichnungen P anstatt $Prec$ und R anstatt Rec. Diese wurden in der vorliegenden Arbeit angepasst.

Betrachtet man die Auswirkungen einer Veränderung des Wertes für β, so erschließt sich daraus die höhere Gewichtung entweder der Genauigkeit oder der Trefferquote. Dafür betrachtet man das F-Maß mit den entsprechenden Formeln für die Genauigkeit und die Trefferquote.

$$
\begin{aligned}
F_\beta &= \frac{(\beta^2+1)Prec\ Rec}{\beta^2 Prec\ +\ Rec} = \frac{(\beta^2+1)*\frac{TP}{TP+FP}*\frac{TP}{TP+FN}}{\beta^2*\frac{TP}{TP+FP}+\frac{TP}{TP+FN}} = \frac{(\beta^2+1)*\frac{TP}{TP+FP}*\frac{TP}{TP+FN}}{\beta^2*\frac{TP*(TP+FN)}{(TP+FP)(TP+FN)}+\frac{TP*(TP+FP)}{(TP+FN)(TP+FP)}} \\
&= \frac{(\beta^2+1)*\frac{TP^2}{(TP+FP)(TP+FN)}}{\frac{\beta^2*TP*(TP+FN)+TP*(TP+FP)}{(TP+FP)(TP+FN)}} \\
&= \frac{(\beta^2+1)*TP^2}{(TP+FP)(TP+FN)} * \frac{(TP+FP)(TP+FN)}{\beta^2*TP*(TP+FN)+TP*(TP+FP)} \\
&= \frac{(\beta^2+1)*TP^2}{\beta^2*TP*(TP+FN)+TP*(TP+FP)} = \frac{(\beta^2+1)*TP^2}{\beta^2*TP^2+\beta^2*FN*TP+TP^2+TP*FP} \\
&= \frac{TP*((\beta^2+1)*TP)}{TP*((\beta^2+1)TP+\beta^2FN+FP)} = \frac{(\beta^2+1)*TP}{(\beta^2+1)TP+\beta^2FN+FP}
\end{aligned}
$$

Jetzt können drei Fälle unterschieden werden:

(a) $\beta = 1$

Ist $\beta = 1$, so erhält man, ausgehend von Formel 2.11, wieder das ursprünglich zu Grunde gelegte harmonische Mittel aus Genauigkeit und Trefferquote.[1] Das bedeutet, weder Genauigkeit noch Trefferquote haben einen größeren Anteil am berechneten Wert für das F-Maß. Sie werden also gleichgewichtet.

(b) $0 \leq \beta < 1$

Wird für β ein Wert größer oder gleich 0, aber kleiner 1 gewählt, so werden die TP im Zähler mit einem Wert größer oder gleich 1, aber kleiner 2 multipliziert. Das Gleiche gilt für die TP im Nenner. Zusätzlich werden die FN im Nenner mit einem Wert größer oder gleich 0, aber kleiner 1 multipliziert. Durch diese Multiplikation wird der Einfluss, den die FN auf den Wert des F-Maßes haben, verringert, während der Einfluss der FP gleich bleibt. Da die FN nur in der Formel zur Berechnung der Trefferquote und nicht in der Formel zur Berechnung der Genauigkeit enthalten sind, wird somit der Einfluss der Trefferquote verringert und damit der Einfluss der Genauigkeit erhöht.

1 Siehe auch die Herleitung im Anhang auf S. 574.

(c) $1 < \beta \leq \infty$
Wird für β ein Wert größer 1, aber kleiner oder gleich ∞ gewählt, so werden die TP im Zähler mit einem Wert größer 2 multipliziert. Das Gleiche gilt für die TP im Nenner. Zusätzlich werden die FN im Nenner mit einem Wert größer 1 multipliziert. Durch diese Multiplikation wird der Einfluss, den die FN auf den Wert des F-Maßes haben, verstärkt, während der Einfluss der FP gleich bleibt. Da die FN nur in der Formel zur Berechnung der Trefferquote und nicht in der Formel zur Berechnung der Genauigkeit enthalten sind, wird somit der Einfluss der Trefferquote erhöht und damit der Einfluss der Genauigkeit verringert.

Das F-Maß kann nicht nur unterschiedlich gewichtet werden, sondern es ist auch möglich, es für die erzeugte Klassifikation unterschiedlich zu berechnen, je nachdem welche Durchschnittsbildung über alle Klassen erfolgen soll. Die TP, FP und FN werden pro Klasse der erzeugten Klassifikation bestimmt. Danach muss das Maß für die gesamte Klassifikation berechnet werden, um die Qualität für alle Klassen vergleichen zu können. Dabei existieren zwei Ansätze[1]:

(a) Microaveraged F-Maß
Microaveraging (deutsch in etwa: kleine - im Sinne von feine - Durchschnittsbildung) wird durch eine Summenbildung über die ermittelten Anzahlen pro Klasse berechnet. Das bedeutet, für jede Klasse werden die oben genannten Werte für TP, FP und FN ermittelt und über alle Klassen addiert. Abschließend wird mit Hilfe dieser Summen der Wert des F-Maßes berechnet. Das F-Maß in seiner microaveraged Form übertreibt dabei die Betonung der Klassen mit den meisten Dokumenten.[2]

(b) Macroaveraged F-Maß
Macroaveraging (deutsch in etwa: große - im Sinne von grobe - Durchschnittsbildung) ist der ungewichtete Durchschnitt über alle Werte des F-Maßes über alle Klassen. Das bedeutet, es wurde für jede Klasse einzeln das F-Maß, wie zuvor beschrieben, berechnet und dann der Durchschnitt über diese Werte aller Klassen bestimmt. In diesem Fall werden die Klassen mit den wenigsten Dokumenten überbetont.[3]

In der vorliegenden Arbeit werden beide Berechnungen zur Evaluation der Klassifizierungsergebnisse eingesetzt.

1 Vgl. Lewis u.a. (2004), S. 381; Lewis (1991), S. 313.
2 Vgl. Echarte u.a. (2011), S. 1677.
3 Vgl. Echarte u.a. (2011), S. 1677.

2.5.3 Evaluation der Clusterergebnisse

2.5.3.1 Internes Qualitätskriterium

Die Zielvorgabe des Clusterns ist, dass möglichst homogene Dokumente in einem Cluster zusammengefasst werden und sich möglichst heterogene Dokumente in verschiedenen Clustern wiederfinden. Ist ein Clustering erzeugt worden, muss die Qualität des Ergebnisses geprüft werden, um festzustellen, ob ein "gutes" Clustering erzeugt wurde.
Ein internes Kriterium für diese Qualität ist bereits in den entsprechenden Clusteralgorithmen versteckt. Es wird jeweils versucht, eine Zielfunktion zu minimieren oder zu maximieren: die Distanz oder die Ähnlichkeit. Für den k-Means-Algorithmus bedeutet das, dass versucht wird, die Summe der quadratischen Distanzen der Feature-Vektoren zu ihrem Clustermittelpunkt - genannt *residual sum of squares (RSS)* - zu minimieren. Dafür wird pro Cluster die Summe über diese quadratischen Distanzen gebildet. Um die quadratische Distanz zu berechnen, wird vom Feature-Vektor $\vec{f}$ der Vektor $\vec{\mu}_{\omega_j}$ des Clustermittelpunkts, in dem sich das Dokument befindet, subtrahiert. Dadurch entsteht wieder ein Vektor. Von diesem wird die Länge ermittelt und diese quadriert. Formal ausgedrückt heißt das[1]:

$$\mathrm{RSS}_{\omega_j} = \sum_{\vec{f} \in \omega_j} |\vec{f} - \vec{\mu}_{\omega_j}|^2 \tag{2.12}$$

Um den RSS-Wert für das gesamte Clustering zu berechnen, wird die Summe über alle RSS_{ω_j} gebildet[2]:

$$\mathrm{RSS} = \sum_{j=1}^{c} \mathrm{RSS}_{\omega_j} \tag{2.13}$$

Unter Berücksichtigung des RSS kann ein Optimierungsproblem definiert werden, dass das Clustering Ω_{min} erzeugen soll, dessen RSS-Wert minimal ist. Dafür wird das Clustering Ω aus der Menge der möglichen Clusterings $\widehat{\Omega}$ gesucht, dessen Summe der quadratischen Distanzen aller Feature-Vektoren zu ihrem jeweiligen Clustermittelpunkt minimal ist.

1 Vgl. Manning u.a. (2008), S. 332.
2 Vgl. Manning u.a. (2008), S. 332.

Das führt zu einem weiteren Stoppkriterium für den k-Means-Algorithmus[1]:

- Der RSS fällt unter einen vorgegebenen Schwellenwert.
  Vorteil: Man erhält eine garantierte Qualität.
  Nachteil: In der Praxis müssen zudem noch die Iterationen begrenzt werden, um Endlosschleifen zu verhindern, die eintreten können, weil die vorgegebene Güte nicht erreichbar ist.

2.5.3.2 Externe Qualitätskriterien

2.5.3.2.1 Rand Index

Der Rand Index[2], als ein Beispiel von vier gebräuchlichen externen Evaluationsmaßen[3] , berechnet den Anteil der vom Clustern erzeugten korrekten Entscheidungen. Dafür wird die Anzahl der korrekten Entscheidungen durch die Gesamtanzahl der Entscheidungen dividiert.[4]

$$\mathrm{RI} = \frac{TP + TN}{TP + FP + FN + TN} \tag{2.14}$$

Ergibt sich ein möglichst hoher Prozentsatz bei Ausdrücken des Maßes in Prozent, so handelt es sich um ein "gutes" Clustering, ein niedriger dagegen verweist auf ein "schlechtes" Clustering. Zu berücksichtigen ist jedoch, dass alle False-negative- und False-positive-Entscheidungen gleich gewichtet werden.[5] Das heißt, es wird nicht unterschieden, ob es nachteiliger ist, ähnliche Dokumente zu trennen, oder unähnliche dem gleichen Cluster zuzuordnen.

2.5.3.2.2 Purity

Ein sehr einfaches Qualitätskriterium ist die so genannte *purity* (deutsch: Reinheitsgrad). Sie kann laut Zhao u.a. (2002)[6] und Strehl u.a. (2000)[7] als die maximal mögliche *precision* für jeden Cluster angegeben werden.

1 Vgl. Manning u.a. (2008), S. 334.
2 Vgl. Rand (1971), S. 847.
3 Vgl. Jain u.a. (1988), S. 174; Dom (2001), S. 9.
4 Die in dieser Arbeit verwendete Formel weicht von der in Rand (1971) verwendeten ab. Eine Herleitung der Formel befindet sich im Anhang C.
5 Diese Aussage trifft ebenfalls auf die True-positive- und True-negative-Entscheidungen zu. Da es sich aber bei diesen Entscheidungen um korrekte Entscheidungen handelt, können sie sich bei einer Gleichgewichtung nicht nachteilig auf das Ergebnis auswirken.
6 Vgl. Zhao u.a. (2002), S. 11.
7 Vgl. Strehl u.a. (2000), S. 61.

Unter *precision* (deutsch: Genauigkeit) wird der Anteil der Dokumente in Cluster ω_j, die zur Klasse L_i gehören, an der Dokumentenanzahl des jeweiligen Clusters verstanden. Also ist der mathematische Ausdruck für die Genauigkeit eines Clusters ω_j: $Prec(L_i, \omega_j) = \frac{h(L_i,\omega_j)}{|\omega_j|}$. Diese Genauigkeit wird für jede Kombination aus diesem Cluster und allen Klassen L_i berechnet und anschließend die maximale Genauigkeit als Purity für den betrachteten Cluster gewählt. Die Gesamtpurity für alle Cluster, also für das Clustering, erhält man über die Summierung der maximalen Genauigkeiten für jeden Cluster. In der hier verwendeten Formel wird zusätzlich die Purity eines einzelnen Clusters mit der jeweiligen relativen Clustergröße - also mit dem Anteil der Anzahl $|\omega_j|$ der Dokumente des betrachteten Clusters ω_j an der Gesamtanzahl $|D|$ der Dokumente - gewichtet, um zu vermeiden, dass kleine Cluster - bspw. mit nur einem Dokument - stärker einfließen als Cluster mit vielen Dokumenten.
Berechnet wird die Purity also folgendermaßen[1]:

$$\text{purity}(\omega_j) = \max_{i \in \{1,\ldots,|L|\}} Prec(L_i, \omega_j) \tag{2.15}$$

$$\text{gewichtete Gesamtpurity} = \sum_{j=1}^{c} \frac{|\omega_j|}{|D|} \text{purity}(\omega_j) \tag{2.16}$$

Die Interpretation des Gesamtmaßes ist folgende[2]: Werte in der Nähe von 0 würden ein "schlechtes" Clustering anzeigen, Werte nahe 1 ein "gutes". Nachteilig an diesem Maß ist die Fokussierung auf die Genauigkeit für jeden einzelnen Cluster, ohne die Trefferquote (englisch: *recall)* einzubeziehen. Diese berechnet den Anteil der ähnlichen Dokumentenpaare, die sich im gleichen Cluster befinden, an der Dokumentenanzahl, die sich eigentlich im gleichen Cluster befinden sollte.[3]

2.5.3.2.3 F-Maß für die Clusterergebnisse

Die beiden in den vorangegangenen Abschnitten angesprochenen Nachteile versucht das so genannte F-Maß[4] zu vermeiden, indem eine Gewichtung angegeben werden kann, ob eher auf die Genauigkeit oder aber auf die Trefferquote Wert gelegt wird. Das F-Maß wurde bereits in Kapitel 2.5.2.2 beschrieben. In Bezug auf ein Clustering ändern sich die Interpretationen der Genauigkeit und der Trefferquote etwas.

1 Vgl. Zhao u.a. (2002), S. 11; Forster (2006), S. 47 f.
2 Das gilt für den hier betrachteten Fall eines harten Clusterings.
3 In Kapitel 2.5.3.2.3 wird sowohl auf die Genauigkeit, als auch auf die Trefferquote weiter eingegangen.
4 Vgl. Jardine u.a. (1971), S. 221.

Wie bereits in Kapitel 2.5.3.2.2 gesagt, versteht man unter der Genauigkeit den Anteil der ähnlichen Dokumentenpaare, die sich im gleichen Cluster befinden, an der Dokumentenanzahl des jeweiligen Clusters. Bezieht man diese Aussage nicht nur auf einen Cluster, sondern auf alle Cluster, also das gesamte Clustering, so versteht man darunter den Anteil der TP an der Summe aus TP und FP. Der Wert, der sich hinter den TP verbirgt, beinhaltet, wie zuvor gesagt, die Anzahl der einander ähnlichen Dokumentenpaare, die dem gleichen Cluster zugeordnet wurden, also die Anzahl der „richtigen“ Zuordnungen von Dokumentenpaaren über alle Cluster. In den Clustern können jedoch auch Dokumente enthalten sein, die nicht in diesen Cluster gehören. Die Anzahl dieser Paare, bei denen der Sachverhalt false positive für mindestens eins der beiden Dokumente gilt, sind die FP. Die Summe aus beiden bildet bezogen auf das gesamte Clustering die Summe der „richtig“ zugeordneten Dokumentenpaare und der „falsch“ zugeordneten Dokumentenpaare über alle Cluster. Die Genauigkeit für das gesamte Clustering ist also wie folgt definiert: $Prec = TP/(TP + FP)$.[1]

Die Trefferquote misst, wie in Kapitel 2.5.3.2.2 gesagt, den Anteil der ähnlichen Dokumentenpaare, die sich im gleichen Cluster befinden, an der Dokumentenanzahl, die sich eigentlich im gleichen Cluster befinden sollte. Bezogen auf das gesamte Clustering bedeutet das den Anteil der TP an der Summe aus TP und FN. Der Wert, der sich hinter den TP verbirgt, beinhaltet, wie zuvor gesagt, die Anzahl der einander ähnlichen Dokumentenpaare, die dem gleichen Cluster zugeordnet wurden, also die Anzahl der „richtigen“ Zuordnungen von Dokumentenpaaren über alle Cluster. In den Clustern können jedoch auch Dokumente fehlen, die eigentlich in diesen Cluster gehören. Die Anzahl dieser Paare sind die FN. Die Summe aus beiden bildet bezogen auf das gesamte Clustering die Summe der „richtig“ zugeordneten Dokumentenpaare und der „fälschlicherweise“ anderen Clustern zugeordneten Dokumentenpaare über alle Cluster. Die Trefferquote für das gesamte Clustering ist also wie folgt definiert: $Rec = TP/(TP + FN)$.
Das F-Maß selbst ist dann das gleiche, wie in Kapitel 2.5.2.2 definiert. Auch der Gewichtungsfaktor im Clustern entspricht dem in der Klassifizierung.

2.5.4 Evaluationsbeispiel

2.5.4.1 Einführung in das Evaluationsbeispiel

Zur Veranschaulichung eines Klassifizierungs- und eines Cluster-Vorgangs und der jeweils anschließenden Evaluation wird anhand eines Beispiels erläutert, wie die Eva-

1 Die Formel unterscheidet sich von der in Kapitel 2.5.3.2.2 für die Genauigkeit eines Clusters in der Purity definierten, da die dort verwendete Formel die Genauigkeit *eines* Clusters berechnet und die hier verwendete Formel die Genauigkeit für das *gesamte Clustering*.

luation durchgeführt wird. Dazu wird eine zufällige Auswahl von 20 Dokumenten aus der Trainingsmenge des RCV1-v2[1] benutzt. Die Klassifikation der 20 Dokumente, wie sie im Goldstandard vorliegt, ist folgender Abbildung zu entnehmen:

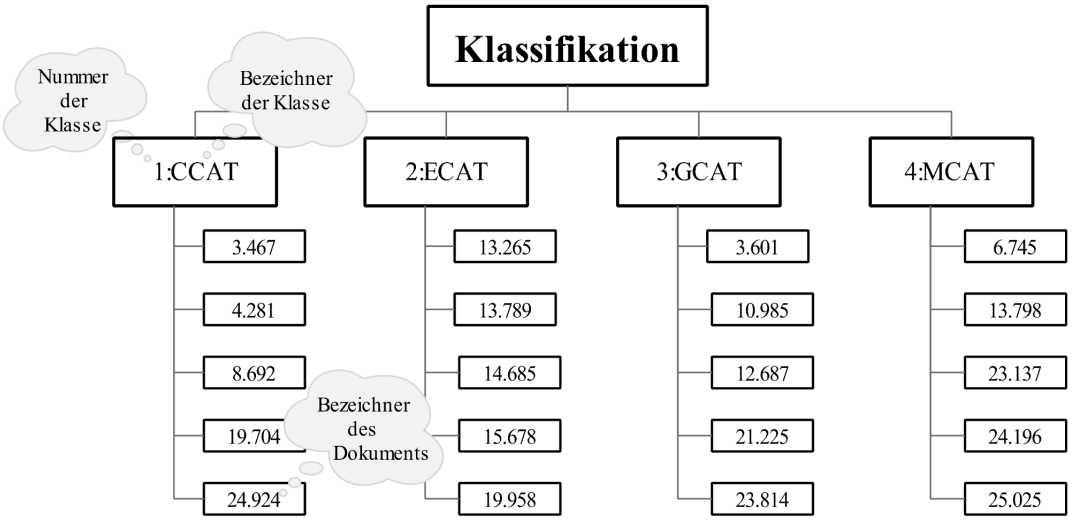


Abbildung 2.7: Goldstandard-Klassifikation von 20 ausgewählten Dokumenten der Reuters-Daten

2.5.4.2 Beispiel für eine Klassifizierung

Ein Klassifizierungsalgorithmus soll diese Klassifikation bestmöglich rekonstruieren.[2] Im Beispiel wird angenommen, dass die in Abbildung 2.8 auf S. 83 zu sehende Klassifikation der Dokumente durch einen Klassifizierungsalgorithmus erzeugt wurde.

Um nun die erzeugte Klassifikation zu evaluieren, muss überprüft werden, welche Dokumente in welcher Klasse der erzeugten Klassifikation enthalten sind und welche Klasse die Dokumente im Goldstandard haben. Aus diesem Vergleich lassen sich die TP, FP und FN, die für die Evaluation benötigt werden, ablesen. Das ist für `Klasse1` in Abbildung 2.9 dargestellt.[3]

1 Das ist der Datensatz, der auch in den Experimenten der vorliegenden Arbeit verwendet wird, siehe Kapitel 4.1.2 und 5.3.2.

2 In einem wirklichen Anwendungsfall und auch in den Experimenten der vorliegenden Arbeit erfolgt eine Einteilung in Trainings- und Testdaten. Da es sich hier nur um ein Beispiel handelt, entfällt dieser Schritt.

3 Eine True-positive-Zuordnung wird durch eine durchgehende Umrandung in grau, eine False-positive-Zuordnung durch eine fein-gestrichelte Umrandung in hellgrau und eine False-negative-Zuordnung durch eine grob-gestrichelte Umrandung in dunkelgrau dargestellt.

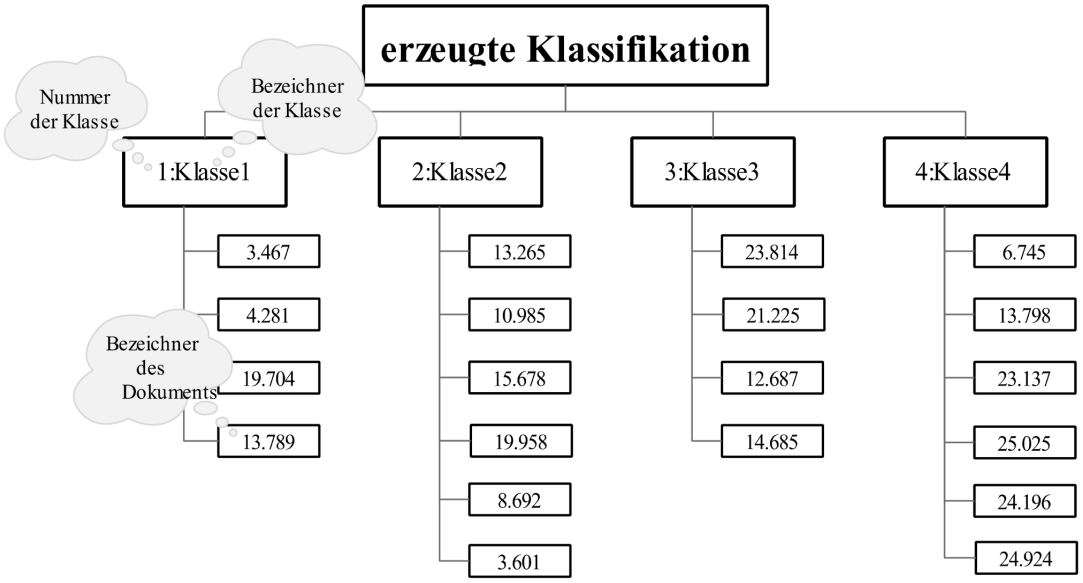


Abbildung 2.8: Klassifikation von 20 ausgewählten Dokumenten der Reuters-Daten durch einen Klassifizierungsalgorithmus

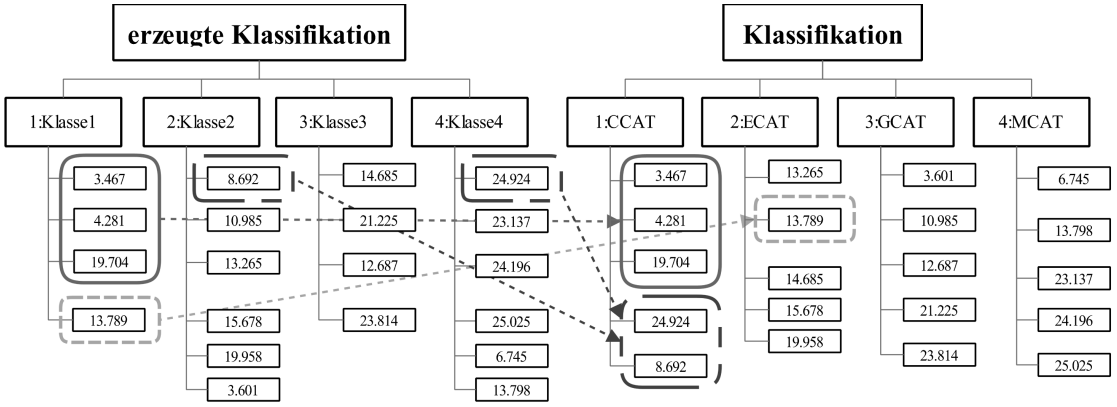


Abbildung 2.9: Vergleich von Klasse1 der erzeugten Klassifikation mit dem Goldstandard

Das bedeutet, für `Klasse1` ergeben sich die folgenden Werte:

	Dokumente	Anzahl
TP	3.467, 4.281, 19.704	3
FP	13.789	1
FN	8.692, 24.924	2

Zusammengefasst für die gesamte erzeugte Klassifikation ergibt sich:

Klasse		Dokumente	Anzahl
1	TP	3.467, 4.281, 19.704	3
	FP	13.789	1
	FN	8.692, 24.924	2
2	TP	13.265, 15.678, 19.958	3
	FP	8.692, 3.601, 10.985	3
	FN	13.789, 14.685	2
3	TP	12.687, 21.225, 23.814	3
	FP	14.685	1
	FN	3.601, 10.985	2
4	TP	6.745, 13.798, 23.137, 24.196, 25.025	5
	FP	24.924	1
	FN		0

Damit ergeben sich in Summe die folgenden Anzahlen:
$TP = 14$; $FP = 6$ und $FN = 6$.

Für den Anteil der richtig klassifizierten Dokumente ergibt sich:

$$\text{korrekter Anteil} = \frac{TP}{n} = \frac{14}{20} = 0,7$$

Für das F_1-Maß im microaveraged-Fall ergibt sich:

$$F_1 = \frac{(1^2+1)PrecRec}{1^2Prec + Rec} = \frac{2 * \frac{TP}{TP+FP} * \frac{TP}{TP+FN}}{1 * \frac{TP}{TP+FP} + \frac{TP}{TP+FN}} = \frac{2 * \frac{14}{14+6} * \frac{14}{14+6}}{1 * \frac{14}{14+6} + \frac{14}{14+6}} = 0,7$$

Für das F_1-Maß im macroaveraged-Fall ergibt sich:

$$\begin{aligned}
F_1 &= \frac{1}{4} * \sum_{i=1}^{4} \frac{(1^2+1)Prec_iRec_i}{1^2Prec_i + Rec_i} = \frac{1}{4} * \sum_{i=1}^{4} \frac{2 * \frac{TP_i}{TP_i+FP_i} * \frac{TP_i}{TP_i+FN_i}}{1 * \frac{TP_i}{TP_i+FP_i} + \frac{TP_i}{TP_i+FN_i}} \\
&= \frac{1}{4} * \left(\frac{2 * \frac{TP_1}{TP_1+FP_1} * \frac{TP_1}{TP_1+FN_1}}{1 * \frac{TP_1}{TP_1+FP_1} + \frac{TP_1}{TP_1+FN_1}} + \frac{2 * \frac{TP_2}{TP_2+FP_2} * \frac{TP_2}{TP_2+FN_2}}{1 * \frac{TP_2}{TP_2+FP_2} + \frac{TP_2}{TP_2+FN_2}} + \frac{2 * \frac{TP_3}{TP_3+FP_3} * \frac{TP_3}{TP_3+FN_3}}{1 * \frac{TP_3}{TP_3+FP_3} + \frac{TP_3}{TP_3+FN_3}} \right. \\
&\quad \left. + \frac{2 * \frac{TP_4}{TP_4+FP_4} * \frac{TP_4}{TP_4+FN_4}}{1 * \frac{TP_4}{TP_4+FP_4} + \frac{TP_4}{TP_4+FN_4}} \right) \\
&= \frac{1}{4} * \left(\frac{2 * \frac{3}{3+1} * \frac{3}{3+2}}{1 * \frac{3}{3+1} + \frac{3}{3+2}} + \frac{2 * \frac{3}{3+3} * \frac{3}{3+2}}{1 * \frac{3}{3+3} + \frac{3}{3+2}} + \frac{2 * \frac{3}{3+1} * \frac{3}{3+2}}{1 * \frac{3}{3+1} + \frac{3}{3+2}} + \frac{0 * \frac{5}{5+1} * \frac{5}{5+0}}{1 * \frac{5}{5+1} + \frac{5}{5+0}} \right) = 0,\overline{69}
\end{aligned}$$

2.5.4.3 Beispiel für ein Clustern

Das Clustern erfolgt hierarchisch[1] im Hinblick auf die vorgegebene Klassifikation der Daten. Die zu Grunde gelegte Klassifikation soll durch die Clusteringalgorithmen „bestmöglich“ erzeugt werden. Dabei betrachtet der GAAC-Algorithmus die Dokumente zunächst jeweils als eigenen Cluster und fügt sie dann schrittweise - also jeweils zwei Dokumente oder ein Dokument und einen bereits erzeugten Cluster - zu einem neuen Cluster zusammen. Das erfolgt so lange, bis das Abbruchkriterium - in diesem Fall die vorher festgelegte Anzahl von 4 Clustern - erreicht ist. Die durchgeführten Schritte lassen sich in einem Dendrogramm abbilden. Zur besseren Lesbarkeit der Abbildung wurden die von Reuters vergebenen Dokumentbezeichner durch kürzere Bezeichner ersetzt. Die Zuordnung von Reuters-Dokumentbezeichnern und in der Abbildung verwendeten Bezeichnern kann der Tabelle 2.1 entnommen werden.

Tabelle 2.1: Zuordnung der Reuters-Dokumentbezeichner zu den im Dendrogramm verwendeten Dokumentbezeichnern

Reuters Dokumentbezeichner	3.467	3.601	4.281	6.745	8.692
hier verwendeter Dokumentbezeichner	T1	T2	T3	T4	T5
Reuters Dokumentbezeichner	10.985	12.687	13.265	13.789	13.798
hier verwendeter Dokumentbezeichner	T6	T7	T8	T9	T10
Reuters Dokumentbezeichner	14.685	15.678	19.704	19.958	21.225
hier verwendeter Dokumentbezeichner	T11	T12	T13	T14	T15
Reuters Dokumentbezeichner	23.137	23.814	24.196	24.924	25.025
hier verwendeter Dokumentbezeichner	T16	T17	T18	T19	T20

Die Abbildung 2.10 auf S. 86 zeigt das Dendrogramm des GAAC-Algorithmus. Das Lesen erfolgt von unten nach oben. Unten - also an den „Blättern“ - sind die einzelnen Dokumente aufgeführt. Ihre Bezeichnung beginnt bei „T1“ und endet bei „T20“. Zu Beginn des Cluster-Vorgangs befinden sich alle Dokumente in einem eigenen Cluster. Dargestellt wird dies durch jeweils eine einzelne gestrichelte vertikale Linie, die von den Bezeichnungen ausgeht.
Werden nun zwei Dokumente zu einem Cluster zusammengefasst, werden sie durch eine durchgezogene horizontale Linie miteinander verbunden. Als Beispiel: im ersten Schritt werden die Dokumente „T9“ und „T10“ zum Cluster „C1“ zusammengefasst. Von diesem erzeugten Cluster führt eine durchgezogene vertikale Linie weiter nach oben, d.h., es handelt sich hier nicht mehr um einzelne Dokumente, sondern um einen

1 Der Grund hierfür ist, dass die Schritte des Algorithmus grafisch durch ein Dendrogramm veranschaulicht werden können.

Cluster, dem einzelne Dokumente hinzugefügt werden können - als Beispiel „T12“ - und der dann durch das Verschmelzen mit diesem neuen Dokument wiederum einen neuen Cluster bildet - hier „C2“.

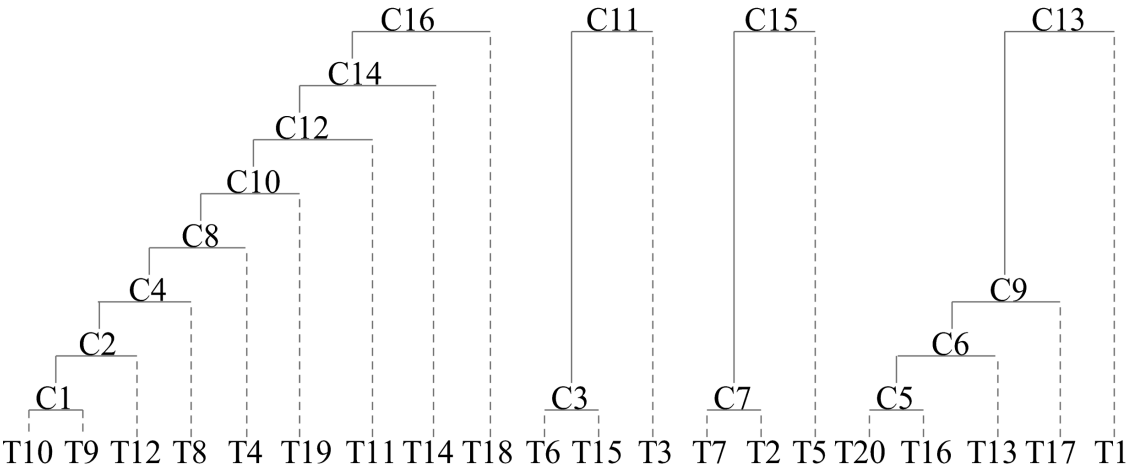


Abbildung 2.10: Beispieldendrogramm für ein hierarchisches Clustering mit 20 Datensätzen

Es existiert auch der Fall, dass zwei Cluster zu einem zusammengefasst werden, jedoch ist dies hier nicht aufgetreten. Der gesamte Ablauf des Algorithmus wird schrittweise aufgezeichnet, bis nur noch vier Cluster vorhanden sind. Auf diese Art und Weise kann der gesamte Ablauf eines Cluster-Vorgangs grafisch dargestellt und nachvollzogen werden.

Das erzeugte Clustering, diesmal nicht als Dendrogramm, sondern als Zuordnung der Dokumente zu den Clustern ohne die Hierarchie der Cluster untereinander, wird in Abbildung 2.11 auf S. 87 dargestellt.[1]

Nachdem der verwendete Algorithmus die Dokumente zu Clustern zusammengefasst hat, muss überprüft werden, inwieweit die erzeugten Cluster die ursprünglichen Klassen abbilden. Dafür wird für jeden Cluster die Anzahl der vorhandenen Dokumente pro Klasse gezählt. In der Abbildung 2.12 auf S. 87 wird für den Cluster `C1` gezeigt, zu welchen Klassen die zugeordneten Dokumente gehören.[2]

1 Zur Berechnung der Evaluationsmaße muss die Hierarchie der Cluster aufgelöst werden, indem nur die 4 Cluster der oberen Ebene betrachtet und die dort enthaltenen Dokumente diesen 4 Clustern direkt zugeordnet werden.

2 Unterschiedliche Farben sowie unterschiedliche Linien der Umrandung stellen unterschiedliche Klassen der Klassifikation dar.

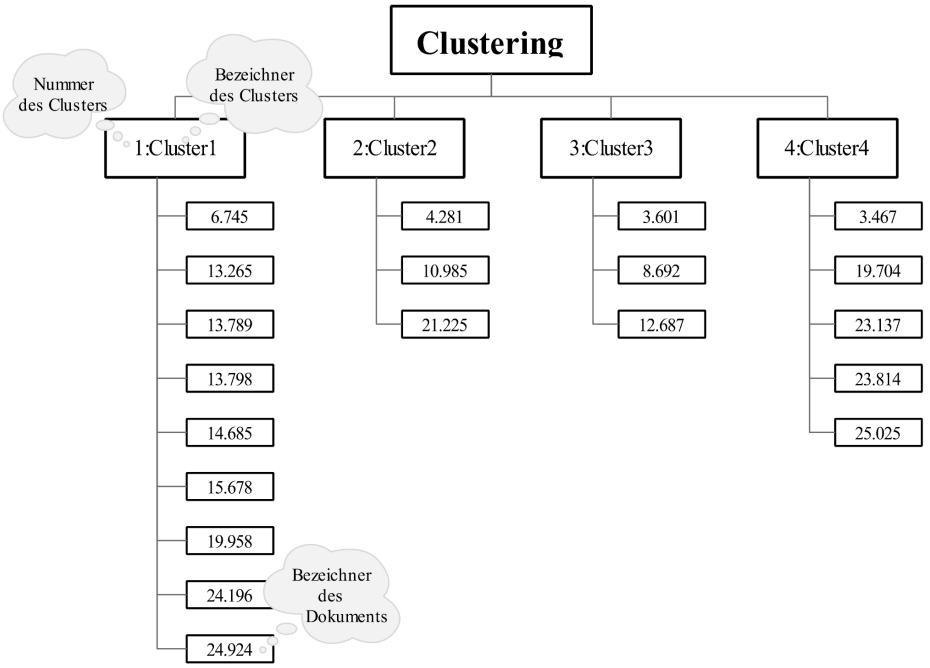


Abbildung 2.11: Darstellung des Clusterings der 20 Datensätze ohne Hierarchie

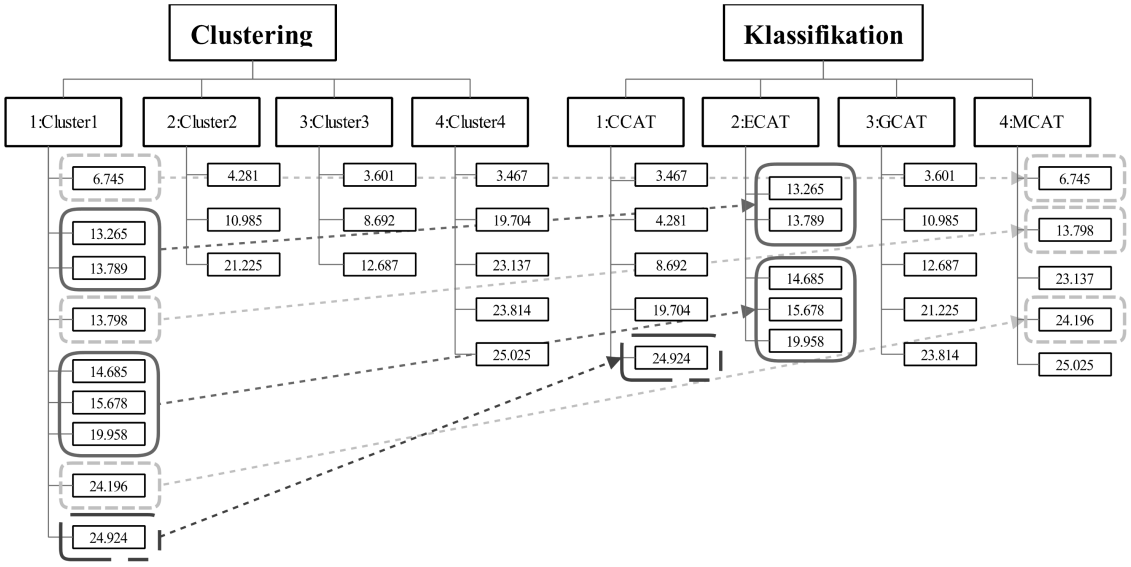


Abbildung 2.12: Vergleich von Cluster1 des erzeugten Clusterings mit dem Goldstandard

Wie in der Abbildung zu sehen ist, gehört ein Dokument zur Klasse „CCAT“, fünf Dokumente gehören zur Klasse „ECAT“ und drei Dokumente zur Klasse „MCAT“. Zählt man diese Zugehörigkeit für alle Dokumente aller Cluster, so erhält man die Werte der Tabelle 2.2 auf S. 88. Die jeweilige Anzahl entspricht $h(L_i, \omega_j)$. Ein konkretes Beispiel für einen solchen Wert ist $h(L_2, \omega_1) = 5$, also die Anzahl der Dokumente in Cluster „C1“, die zur Klasse „2“, dabei handelt es sich um „ECAT“, gehören, ist fünf.

Tabelle 2.2: Anzahl der Dokumente in Cluster ω_j, die zur Klasse L_i gehören

	L_i				$\lvert\omega_j\rvert$
	CCAT	ECAT	GCAT	MCAT	
ω_j	C1	1	5	0	3	9
	C2	1	0	2	0	3
	C3	1	0	2	0	3
	C4	2	0	1	2	5
	$\lvert L_i\rvert$	5	5	5	5	20

Mit Hilfe dieser Tabelle und der Sachverhalte (a) bis (d) auf S. 74 können die dort definierten Werte berechnet werden. Das bedeutet, für die TP ergibt sich im Beispiel:

$$
\begin{aligned}
TP = \sum_{i=1}^{|L|}\sum_{j=1}^{c}\binom{h(L_i,\omega_j)}{2} &= \binom{h(L_1,\omega_1)}{2} + \binom{h(L_1,\omega_2)}{2} + \binom{h(L_1,\omega_3)}{2} + \binom{h(L_1,\omega_4)}{2} + \\
&\binom{h(L_2,\omega_1)}{2} + \binom{h(L_2,\omega_2)}{2} + \binom{h(L_2,\omega_3)}{2} + \binom{h(L_2,\omega_4)}{2} + \binom{h(L_3,\omega_1)}{2} + \binom{h(L_3,\omega_2)}{2} \\
&+ \binom{h(L_3,\omega_3)}{2} + \binom{h(L_3,\omega_4)}{2} + \binom{h(L_4,\omega_1)}{2} + \binom{h(L_4,\omega_2)}{2} + \binom{h(L_4,\omega_3)}{2} + \binom{h(L_4,\omega_4)}{2} \\
&= \binom{1}{2} + \binom{1}{2} + \binom{1}{2} + \binom{2}{2} + \binom{5}{2} + \binom{0}{2} + \binom{0}{2} + \binom{0}{2} + \binom{0}{2} + \binom{2}{2} + \binom{2}{2} + \binom{1}{2} + \binom{3}{2} \\
&+ \binom{0}{2} + \binom{0}{2} + \binom{2}{2} \\
&= 0+0+0+1+10+0+0+0+0+1+1+0+3+0+0+1 = 17
\end{aligned}
$$

Als FN erhält man:

$$
\begin{aligned}
FN = \sum_{i=1}^{l}\binom{|L_i|}{2} - TP &= \binom{|L_1|}{2} + \binom{|L_2|}{2} + \binom{|L_3|}{2} + \binom{|L_4|}{2} - TP = 4 * \binom{5}{2} - TP \\
&= 4 * 10 - 17 = 23
\end{aligned}
$$

Für FP ergibt sich:

$$FP = \sum_{j=1}^{k} \binom{|\omega_j|}{2} - TP = \binom{|\omega_1|}{2} + \binom{|\omega_2|}{2} + \binom{|\omega_3|}{2} + \binom{|\omega_4|}{2} - TP$$
$$= \binom{9}{2} + \binom{3}{2} + \binom{3}{2} + \binom{5}{2} - TP = 36 + 3 + 3 + 10 - 17 = 35$$

Abschließend ergibt sich für die TN:

$$TN = \binom{|D|}{2} - (TP + FN + FP) = \binom{20}{2} - (17 + 23 + 35) = 190 - 75 = 115$$

Unter Verwendung dieser Anzahlen können die Werte der Evaluationsmaße berechnet werden. Für den Rand Index ergibt sich das folgende Ergebnis für das Clustering des Beispiels.

$$RI = \frac{TP + TN}{TP + FP + FN + TN} = \frac{17 + 115}{17 + 35 + 23 + 115} = \frac{132}{190} = 0,694736842$$

Für die Purity müssen zunächst die Genauigkeiten für jeden Cluster bestimmt werden. Das erfolgt, indem für jeden Cluster alle Klassen betrachtet und die Ergebnisse der Formel $Prec(L_i, \omega_j) = \frac{h(L_i,\omega_j)}{|\omega_j|}$ aufgelistet werden. Sie befinden sich in den Zellen der Tabelle 2.3. Zusätzlich beinhaltet die Tabelle in der letzten Spalte bereits den maximalen Wert für die Genauigkeit des jeweiligen Clusters, also die Purity für den Cluster. Die dunkel eingefärbten Zellen der Tabelle verdeutlichen, welche Klasse am entsprechenden Wert beteiligt ist.

Tabelle 2.3: Darstellung der $Prec(L_i, \omega_j)$-Werte pro Cluster mit Angabe des jeweiligen Maximalwertes

		L_i				
ω_j		L_1	L_2	L_3	L_4	$\max_{i \in \{1,\ldots,\lvert L\rvert\}} Prec(L_i, \omega_j) = \text{purity}(\omega_j)$
	ω_1	$\frac{1}{9}$	$\frac{5}{9}$	$\frac{0}{9}$	$\frac{3}{9}$	$\frac{5}{9}$
	ω_2	$\frac{1}{3}$	$\frac{0}{3}$	$\frac{2}{3}$	$\frac{0}{3}$	$\frac{2}{3}$
	ω_3	$\frac{1}{3}$	$\frac{0}{3}$	$\frac{2}{3}$	$\frac{0}{3}$	$\frac{2}{3}$
	ω_4	$\frac{2}{5}$	$\frac{0}{5}$	$\frac{1}{5}$	$\frac{2}{5}$	$\frac{2}{5}$

Die Berechnung der gewichteten Gesamtpurity sieht wie folgt aus:

$$\begin{aligned}\text{gewichtete Gesamtpurity} = \sum_{j=1}^{c} \frac{|\omega_j|}{n}\text{purity}(\omega_j) = \frac{|\omega_1|}{20}\text{purity}(\omega_1) + \frac{|\omega_2|}{20}\text{purity}(\omega_2) + \frac{|\omega_3|}{20}\text{purity}(\omega_3) + \\ \frac{|\omega_4|}{20}\text{purity}(\omega_4) = \frac{9}{20} * \frac{5}{9} + \frac{3}{20} * \frac{2}{3} + \frac{3}{20} * \frac{2}{3} + \frac{5}{20} * \frac{2}{5} = \frac{11}{20} \\ = 0,55\end{aligned}$$

Zur Berechnung des Wertes des F-Maßes können die TP, FP und FN benutzt werden. Die Formel für das F-Maß lautet:

$$F_\beta = \frac{(\beta^2 + 1) * Prec\ Rec}{\beta^2 * Prec + \ Rec} = \frac{(\beta^2 + 1) * \frac{TP}{TP+FP} * \frac{TP}{TP+FN}}{\beta^2 * \frac{TP}{TP+FP} + \frac{TP}{TP+FN}}$$

Damit ergibt sich für das Beispiel:

- für $\beta = 1$

$$F_1 = \frac{(1^2 + 1) * \frac{17}{17+35} * \frac{17}{17+23}}{1^2 * \frac{17}{17+35} + \frac{17}{17+23}} = \frac{2 * \frac{17}{52} * \frac{17}{40}}{1 * \frac{17}{52} + \frac{17}{40}} = 0,3696$$

- für $\beta = 0,5$[1]

$$F_{0,5} = \frac{(0,5^2 + 1) * \frac{17}{17+35} * \frac{17}{17+23}}{0,5^2 * \frac{17}{17+35} + \frac{17}{17+23}} = \frac{1,25 * \frac{17}{52} * \frac{17}{40}}{0,25 * \frac{17}{52} + \frac{17}{40}} = 0,3427$$

- für $\beta = 2$[2]

$$F_2 = \frac{(2^2 + 1) * \frac{17}{17+35} * \frac{17}{17+23}}{2^2 * \frac{17}{17+35} + \frac{17}{17+23}} = \frac{5 * \frac{17}{52} * \frac{17}{40}}{4 * \frac{17}{52} + \frac{17}{40}} = 0,4009$$

1 Hiermit erfolgt eine höhere Gewichtung der Genauigkeit.
2 Hiermit erfolgt eine höhere Gewichtung der Trefferquote.

3 Vorgehensweise zur Ermittlung von Text-Suffix-Fragment-Features

3.1 Erläuterung der Grundlagen von Text-Suffix-Fragment-Features

3.1.1 Grundlegende Datenstrukturen

3.1.1.1 Datenstruktur Array

Die Datenstruktur *Array* wird auch als *Feld* bezeichnet.[1] Es ist wie folgt definiert:

Definition 2. *Bei einem* ***Array*** *handelt es sich um ein Datenobjekt, das eine* ***Menge von Elementen*** *mit dem* ***gleichen Datentyp*** *enthält. Das Array hat eine* ***feste Größe****.*

So kann ein Array beispielsweise ganze Zahlen enthalten oder Zeichenketten oder auch selbst definierte Datentypen. Es dient dazu, viele Variablen gleichen Typs deklarieren und verwalten zu können, so dass der Zusammenhang zwischen diesen Variablen ersichtlich ist.[2]
Der Zugriff auf die Elemente des Arrays erfolgt über einen Index. Das bedeutet, wenn man den Index des gesuchten Elements kennt, ist es nicht nötig, die ganze Datenstruktur zu durchsuchen, bis man das Element gefunden hat, sondern kann direkt auf das Element zugreifen. Ausgedrückt wird das folgendermaßen: $a[i]$ bedeutet, dass das Element an Position i des Arrays a zurückgeliefert wird. Die eckigen Klammern werden als „*Indexoperator*“[3] bezeichnet.
Unterschieden werden zudem *eindimensionale* und *mehrdimensionale* Arrays. Im ersten Fall entspricht ein solches Array einem Vektor aus der Mathematik, im zweiten Fall einer Matrix.[4] Das Zugriffsbeispiel weiter oben zeigt den Zugriff auf ein eindimensionales Array. Der Zugriff auf ein zweidimensionales Array könnte wie folgt aussehen: $a[i][j]$. Das würde beispielsweise bedeuten, dass das Element in Zeile i und Spalte j des Arrays a zurückgeliefert wird.

1 Vgl. bspw. Hubwieser u.a. (2004), S. 59; Pomberger u.a. (2008), S. 126; Blum (2004), S. 4.
2 Vgl. Balzert (1999), S. 383.
3 Pomberger u.a. (2008), S. 127.
4 Vgl. Pomberger u.a. (2008), S. 127.

3.1.1.2 Datenstruktur Tree

Ein *Tree* oder auch *Baum* ist folgendermaßen definiert[1]:

Definition 3. *Ein Baum besteht aus einer endlichen Menge von* ***Knoten****. Diese haben eine endliche, begrenzte Anzahl von* ***Nachfolgern****, auch* ***Söhne*** *genannt. Es existiert ein besonderer Knoten im Baum: die* ***Wurzel****. Dieser Knoten hat als einziger Knoten im Baum keinen* ***Vorgänger****. Ein Vorgänger eines anderen Knotens wird als der* ***Vater*** *des anderen Knotens bezeichnet. Zwei Knoten p und q sind durch eine* ***Kante*** *miteinander verbunden, wenn Knoten q der Sohn von Knoten p ist und demnach Knoten p der Vater von Knoten q ist. Es gibt Knoten, die keine Nachfolger haben, diese Knoten nennt man* ***Blätter****. Alle anderen Knoten werden als* ***innere Knoten*** *bezeichnet. Jeder Knoten eines Baumes bildet einen* ***Teilbaum*** *des Gesamtbaumes mit diesem Knoten als Wurzel.*

3.1.2 Definition Text-Suffix

Ein Suffix ist laut Duden ein

> *„... hinten an den Wortstamm angefügtes Wortbildungselement ...“*[2]

Übertragen auf einen natürlichsprachlichen Text bedeutet das in der vorliegenden Arbeit, dass es sich um eine Sequenz von Zeichen handelt, die sich am Ende dieses Textes befindet.
Wie bereits in Kapitel 1.5 erläutert, lautet die Definition eines Text-Suffix in dieser Arbeit wie folgt[3]:

Definition 4. *Ein* ***Text*** *T ist eine geordnete Abfolge von Zeichen und wird als* ***eine*** *Zeichenkette verstanden. Das bedeutet,* ***alle Zeichen*** *dieses Textes werden von 1 bis zur Länge des Textes $|T|$* ***aufsteigend durchnummeriert****. Leerzeichen werden dabei ebenfalls mit einer entsprechenden Nummer versehen. Diese Nummer bezeichnet die eindeutige* ***Position*** *des jeweiligen Zeichens im Text.*

Definition 5. *Für jeden Text T heißt $T[i..j]$ der* ***Teiltext*** *von T, der an der Position i beginnt und an der Position j endet.*

Definition 6. *$T[i..|T|]$ ist das* ***Suffix*** *des Textes T, das an Position i beginnt. Es wird in dieser Arbeit als* ***Text-Suffix*** *bezeichnet. Als verkürzte Schreibweise wird $T[i]$ eingeführt.*

1 Vgl. Ottmann u.a. (2012), S. 259 f.
2 Scholze-Stubenrecht u.a. (2006), S. 985, Kursivschreibung durch die Verfasserin.
3 Vgl. Gusfield (1999), S. 3 f.; Kim u.a. (2003), S. 187 f.; Manber u.a. (1990), S. 320; Gonnet u.a. (1992), S. 67.

Ein Präfix dagegen ist ein *„... vorn an den Wortstamm angefügtes Wortbildungselement ...“*[1].

Definition 7. *In der vorliegenden Arbeit ist $T[1..i]$ das **Präfix** des Textes T, das an Position i endet. Es wird als **Text-Präfix** bezeichnet.*

In Abbildung 3.1 wird anhand eines Beispiels verdeutlicht, was ein Text-Präfix und ein Text-Suffix ist.

Abbildung 3.1: Beliebige Darstellung eines Text-Präfixes und eines Text-Suffixes

3.1.3 Suffix Tree

3.1.3.1 Definition eines Suffix Trees

Das Überprüfen, ob Features in einem Text vorkommen, spielt eine wichtige Rolle im Rahmen der Klassifzierung oder des Clusterns dieses Textes, siehe Kapitel 2. Dieses Überprüfen entspricht der Aufgabe, ein *Muster* (engl. *Pattern*) in einem Text zu finden. Dabei ist die Aufgabenstellung in der vorliegenden Arbeit so, dass es eine potenziell große Menge an Mustern in Texten zu suchen gilt. Aus diesem Grund werden die Texte zunächst aufbereitet, das ist die so genannte *Vorbereitung* (engl. *preprocessing*).[2] Die Vorbereitung erfolgt so, dass später beim eigentlichen Suchen des Musters im Text Aufwand gespart werden kann. Eine Möglichkeit, die Texte aufzubereiten, besteht darin, sie in der Datenstruktur Suffix Tree zu speichern.[3]

1 Scholze-Stubenrecht u.a. (2006), S. 802, Kursivschreibung durch die Verfasserin.
2 Vgl. Gusfield (1999), S. 6.
3 Die Datenstruktur Suffix Tree wurde zunächst als *prefix tree* von Weiner (1973), S. 3, eingeführt und später auch als *position tree* bezeichnet, vgl. Aho u.a. (1974), S. 346 f.; Majster u.a. (1979), S. 191. Eine weitere Datenstruktur, die dem Suffix Tree sehr ähnlich ist, ist der *PAT Tree*, vgl. Gonnet u.a. (1992), S. 68-70. Die PAT-Datenstruktur wurde bereits in Gonnet (1983), 121 f., beschrieben. In Kombination mit einem *Patricia tree*, vgl. Morrison (1968) für die PATRICIA-Datenstruktur und vgl. Gonnet u.a. (1991), S. 140-143, für Patricia Trees, wird sie zu einem PAT Tree.

Ein Suffix Tree ist wie folgt definiert[1]:

Definition 8. *Ein Suffix Tree einer Zeichenkette ist ein Baum, der alle Suffixe dieser Zeichenkette enthält. Eine solche Zeichenkette ist in der vorliegenden Arbeit ein natürlichsprachlicher Text T der Länge $|T|$ aus einem Dokument.*
Folgende Eigenschaften gelten für diesen Baum:

1. *Der Baum besteht aus einer Wurzel und exakt $|T| + 1$ Blättern sowie einer nicht näher bestimmten Anzahl an inneren Knoten.*
2. *Die Kanten zu den Blättern sind mit jeweils einem Suffix des Textes beschriftet.*
3. *Das Suffix, das an Blatt i endet, beginnt an Position i des Textes.*
4. *Keine zwei Kantenbeschriftungen des Baumes beginnen im gleichen Knoten mit dem gleichen Zeichen.*

3.1.3.2 Aufbau eines Suffix Trees

Die „naive“[2] Vorgehensweise zum Aufbau des Suffix Trees besteht darin, jedes Suffix des Textes in den Baum einzufügen. Zunächst wird das „$“ als eindeutiges Zeichen für das Ende des Textes festgelegt.[3]
Der Text T wird zunächst um das „$“ ergänzt.[4] Man beginnt mit dem ersten Suffix und fügt diesen in den Baum ein, indem man von der Wurzel eine Kante mit dem ersten Suffix als Beschriftung einfügt. Anschließend fährt man, unter Beachtung der oben erläuterten Einschränkungen fort, bis alle Suffixe des Textes im Suffix Tree enthalten sind.

1 Vgl. Gusfield (1999), S. 90 f.; Abouelhoda u.a. (2004), S. 56.
2 „Naiv“ dient hier als Bezeichnung für eine erste intuitive Vorgehensweise, die jedoch meist weder im Hinblick auf die benötigte Zeit noch auf den benötigten Speicherplatz die bestmögliche Vorgehensweise darstellt.
3 Vgl. Gusfield (1999), S. 91; Ko u.a. (2003), S. 202. Das Verwenden dieses Zeichens als eindeutiges Endezeichen ist nur so lange sinnvoll, wie das Zeichen selbst im Text nicht auftaucht. Ein eindeutiges Endezeichen ist aber nötig, vgl. McCreight (1976), S. 263. Es dient dazu, die Suffixe voneinander unterscheiden zu können, so dass nur ein Vergleich eines Suffixes mit sich selbst vollständige Übereinstimmung ergibt, während der Vergleich eines Suffixes mit allen anderen Suffixen des Textes spätestens aufgrund des Endezeichens keine vollständige Übereinstimmung ergibt, vgl. Gonnet u.a. (1992), S. 68. Anders ausgedrückt: Ist ein Suffix ein Präfix eines anderen Suffixes des Textes und gäbe es kein Endezeichen, so würde das Suffix, das Präfix des anderen Suffixes ist, im Suffix Tree auf einer Kante zu einem inneren Knoten stehen, siehe Algorithmus 5. Das hieße, dass es nicht in einem Blattknoten enden würde und deshalb die Eigenschaft 1 der Definition 8 verletzt werden würde.
4 Die Länge des Textes wird dennoch mit $|T|$ bezeichnet, sie entspricht nur nicht mehr der Länge von T ohne das Endezeichen.

Ein Algorithmus, der auf der Beschreibung von Gusfield (1999)[1] basiert, ist im Folgenden dargestellt:

Algorithmus 5 Algorithmus des naiven Aufbaus eines Suffix Trees

Eingabe: Text T mit Länge $|T|$

Ausgabe: Suffix Tree des Textes T

1: Erzeuge die Wurzel r des Suffix Tree ST_1
2: Erzeuge einen Blattknoten mit Beschriftung 1
3: Erzeuge die erste Kante des Suffix Tree ST_1 zwischen der Wurzel und dem Blattknoten mit dem Suffix $T[1]$ als Beschriftung
4: **for** $i = 2, ..., |T|$ **do**
5: Suche, ausgehend von der Wurzel, eine Kante in ST_{i-1}, deren Beschriftung mit dem ersten Zeichen von $T[i]$ beginnt
6: **if** Kante ist vorhanden **then**
7: Vergleiche zeichenweise die Beschriftung der aktuellen Kante (w, v) mit $T[i]$
8: **if** keine Übereinstimmung zwischen Beschriftung von (w, v) und $T[i]$ **then**
9: **if** aktuelle Position ist auf (w, v), aber nicht in einem Knoten **then**
10: Erzeuge auf der Kante zwischen dem letzten Zeichen, das übereinstimmte, und dem ersten, das nicht übereinstimmte, einen Knoten u
11: Beschrifte die neue Kante (w, u) mit dem Teil des Suffix $T[i]$ der übereinstimmte
12: Beschrifte die neue Kante (u, v) mit dem übrigbleibenden Teil der ursprünglichen Beschriftung der Kante (w, v)
13: **else**
14: aktuelle Position ist Knoten u
15: Erzeuge einen neuen Blattknoten mit Beschriftung i
16: Erzeuge eine Kante (u, i) vom Knoten w zum Blattknoten i und beschrifte sie mit dem nicht-übereinstimmenden Teil von Suffix $T[i]$
17: **end if**
18: **end if**
19: **else**
20: aktuelle Position ist Knoten r, die Wurzel des Suffix Trees ST_i

1 Vgl. Gusfield (1999), S. 93.

Algorithmus 5 Algorithmus des naiven Aufbaus eines Suffix Trees (Teil 2)

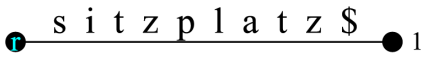
```
21:         Erzeuge einen neuen Blattknoten mit Beschriftung i
22:         Erzeuge eine Kante (r, i) vom Knoten r zum Blattknoten i und beschrifte
            sie mit dem Suffix T[i..|T|]
23:     end if
24: end for
```

Ein Beispiel für den naiven Aufbau eines Suffix Trees für den Text T =„sitzplatz“ sieht wie folgt aus[1]:

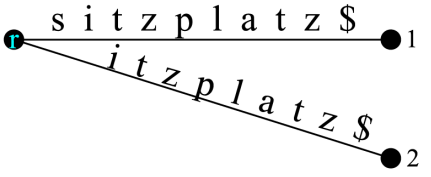
Ergänze den Text T um das Endezeichen „$“.

T = „sitzplatz$“

Erzeuge die Wurzel r des Suffix Tree ST_1, den Blattknoten 1 und die Kante $(r, 1)$ mit der Beschriftung „sitzplatz$“.

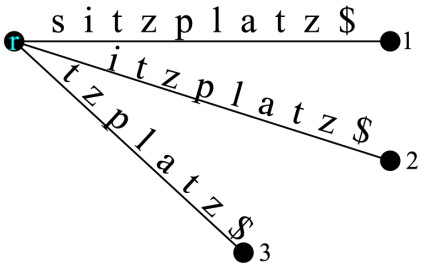


Das nächste Suffix ist „itzplatz$“. Da von der Wurzel ausgehend noch keine Kante vorhanden ist, deren Beschriftung mit „i“ beginnt, muss von der aktuellen Position, Knoten r, der Wurzel, eine neue Kante zwischen r und dem neuen Blattknoten $i = 2$ mit der Beschriftung „itzplatz$“ erzeugt werden.

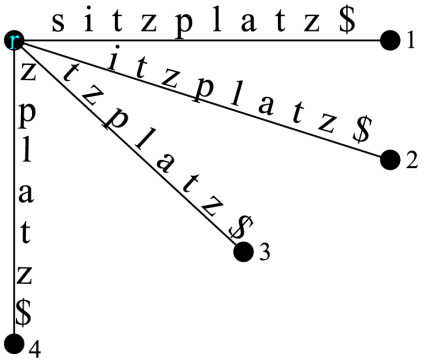


1 Das Symbol „=“ bedeutet hier und im Folgenden, dass eine Zuweisung stattfindet. Beispielsweise würde $x = y$ bedeuten, dass die Variable x den Wert y oder den Wert der Variablen y erhält. Das dient zur Unterscheidung zwischen einer Zuweisung und der Prüfung auf die Gleichheit von Werten. Die Prüfung auf die Gleichheit von Werten wird mit einem „==“ dargestellt.

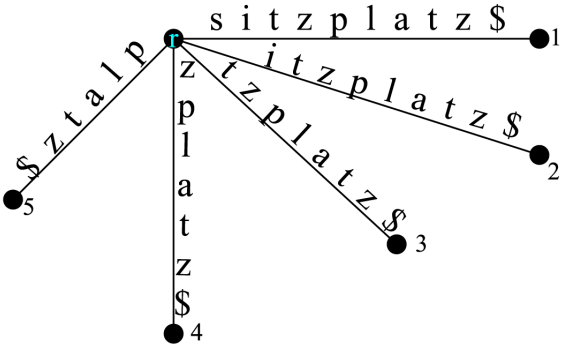
Das nächste Suffix ist „tzplatz$“. Da von der Wurzel ausgehend noch keine Kante vorhanden ist, deren Beschriftung mit „t“ beginnt, muss von der aktuellen Position, Knoten r, der Wurzel, eine neue Kante zwischen r und dem neuen Blattknoten $i = 3$ mit der Beschriftung „tzplatz$“ erzeugt werden.



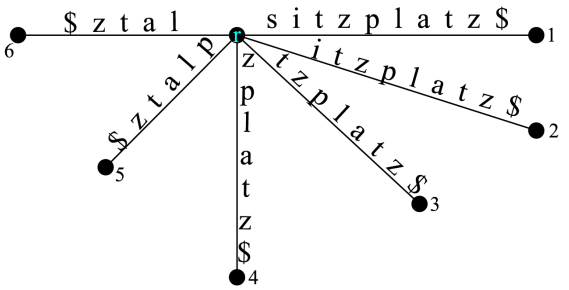
Für das nächste Suffix „zplatz$“ gilt das vorher Gesagte, da von der Wurzel ausgehend noch keine Kante vorhanden ist, deren Beschriftung mit „z“ beginnt. Es wird also eine neue Kante zwischen r und dem neuen Blattknoten $i = 4$ mit der Beschriftung „zplatz$“ erzeugt.



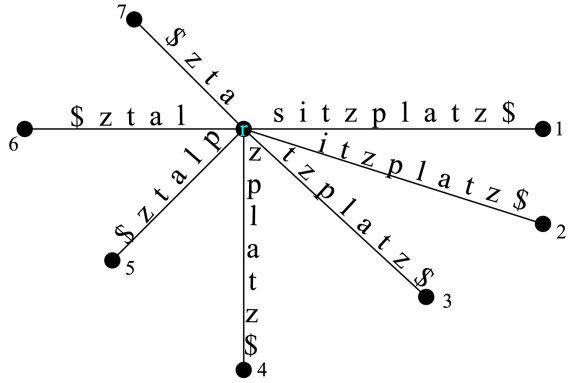
Für das nächste Suffix „platz$“ gilt das vorher Gesagte, da von der Wurzel ausgehend noch keine Kante vorhanden ist, deren Beschriftung mit „p“ beginnt. Es wird also eine neue Kante zwischen r und dem neuen Blattknoten $i = 5$ mit der Beschriftung „platz$“ erzeugt.



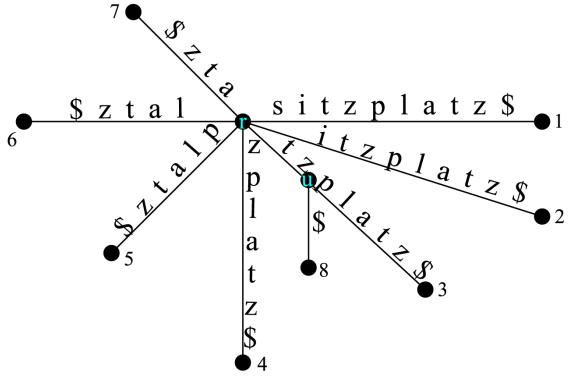
Auch für das Suffix „latz$“ ist von der Wurzel ausgehend noch keine Kante vorhanden, deren Beschriftung mit „l“ beginnt. Es wird also eine neue Kante zwischen r und dem neuen Blattknoten $i = 6$ mit der Beschriftung „latz$“ erzeugt.



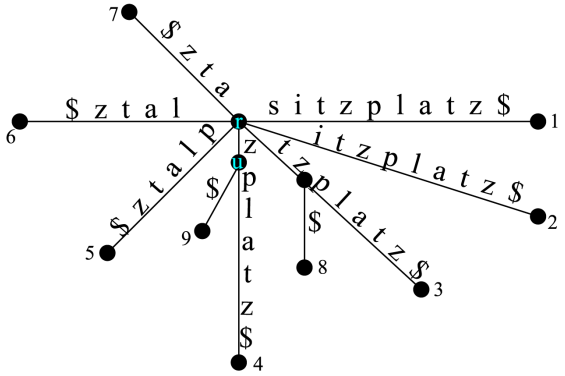
Für das Suffix „atz$“ ist von der Wurzel ausgehend noch keine Kante vorhanden, deren Beschriftung mit „a“ beginnt. Es wird also eine neue Kante zwischen r und dem neuen Blattknoten $i = 7$ mit der Beschriftung „atz$“ erzeugt.



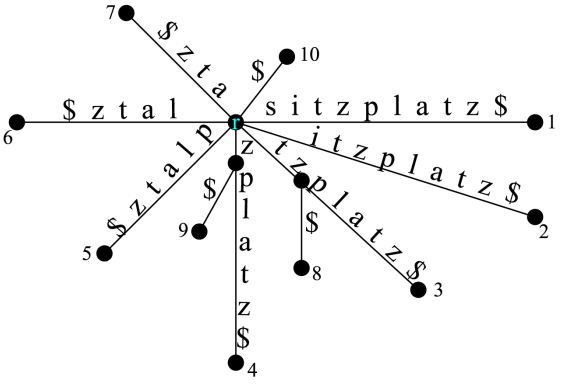
Für das Suffix „tz$“ ist von der Wurzel ausgehend eine Kante vorhanden, deren Beschriftung mit „tz“ beginnt. Es wird also ein neuer Knoten u nach dem „tz“ auf der Kante zum Blattknoten 3 erzeugt. Davon ausgehend wird eine neue Kante zwischen u und dem neuen Blattknoten $i = 8$ mit der Beschriftung „$“ erzeugt, da das Präfix „tz“ bereits im Baum vorhanden ist.



Für das Suffix „z$“ ist ebenfalls von der Wurzel ausgehend eine Kante vorhanden, deren Beschriftung mit „z“ beginnt. Es wird also ein neuer Knoten u nach dem „z“ auf der Kante zum Blattknoten 4 erzeugt. Davon ausgehend wird eine neue Kante zwischen u und dem neuen Blattknoten $i = 9$ mit der Beschriftung „$“ erzeugt, da das Präfix „z“ bereits im Baum vorhanden ist.



Für das letzte Suffix "$“ ist keine Kante vorhanden. Es wird also ein neuer Blattknoten 10 erzeugt und von der Wurzel eine neue Kante zwischen r, und dem neuen Blattknoten mit der Beschriftung „$“ erzeugt.



Anhand dieser Vorgehensweise lässt sich ein Suffix Tree für einen Text aufbauen. Im Beispiel wurde nur ein Wort als Text verwendet, da die Darstellung sonst unübersichtlich geworden wäre. Das Prinzip beim Aufbau ist jedoch das gleiche, nur dass dann auch Leerzeichen aufgenommen werden müssen.
Dieser naive Aufbau ist jedoch nicht effizient im Hinblick auf den verbrauchten Speicherplatz und die benötigte Laufzeit zum Aufbau des Suffix Trees. Daher existieren verschiedene Algorithmen zur Verbesserung dieser Nachteile, die jedoch nicht im Fokus der vorliegenden Arbeit stehen.[1]

3.1.3.3 Anwendungsmöglichkeiten für Sufix Trees

Anhand des oben gegebenen Beispiels wird nur zu einem Teil deutlich, wofür Suffix Trees verwendet werden können. Von Gusfield (1999)[2] vorgeschlagene Anwendungs-

1 Für einen Überblick über solche Algorithmen vgl. bspw. Gusfield (1999), S. 94-119; Ukkonen (1995); McCreight (1976); Weiner (1973); Giegerich u.a. (1999); Tata u.a. (2004); Kurtz (1999).
2 Vgl. bspw. Gusfield (1999), S. 122-149, 156-168, 196-207.

möglichkeiten, die auf Texte übertragen werden können und in Bezug auf Klassifizierung und Clustern natürlichsprachlicher Texte eine Rolle spielen können[1], sind zum Beispiel:

- **Suchen nach exakten Übereinstimmungen mit einer vorgegebenen Zeichenkette**
  Verfügt man über einen Suffix Tree, der einen bestimmten Text komplett darstellt, und sucht man in diesem Text nach einer Zeichenkette, besitzt der Baum bereits für das erste Zeichen der Zeichenkette einen eindeutigen Startpunkt. Von diesem Startpunkt ausgehend wird die Zeichenkette durch einfaches Absteigen im Baum gefunden. Zusätzlich zu der Information, dass in dem Text eine solche Zeichenkette existiert, erhält man auch die Information an welcher Stelle. Das ist durch die Knotenbezeichnung möglich. Wird das erste Zeichen dagegen nicht gefunden, so benötigt man keine weitere Suche, da dann die Zeichenkette nicht im Baum existiert und somit auch nicht im Text auftauchen kann.

- **Suchen nach teilweisen Übereinstimmungen mit einer vorgegebenen Zeichenkette**
  Es können nicht nur exakte Übereinstimmungen gefunden werden, sondern auch Teilzeichenketten der gesuchten Zeichenkette. Dafür sucht man einen Einstiegspunkt, der dann auch mitten in der Zeichenkette liegen kann, und verfolgt den entsprechenden Pfad im Baum, bis sich keine Übereinstimmungen mehr finden lassen. Diese Position kann man sich merken und durch das Betrachten aller nachfolgenden Blätter die genaue Position oder die genauen Positionen der Teilzeichenkette im Text ermitteln.

- **Suchen nach allen Vorkommen eines bestimmten Musters oder Suchen nach allen Suffixen, die ein bestimmtes Muster enthalten**
  Beide genannten Suchmöglichkeiten sind im Prinzip gleich. Vorgegeben wird ein bestimmtes Muster und man sucht dieses Muster im Baum. Durch das Zählen der nachfolgenden Blätter würde man die Anzahl des Musters im Text ermitteln. Verfolgt man die Pfade bis zu den Blättern, weiß man, welche Suffixe dieses Muster enthalten, und kann sie angeben. Das wird durch die Knotenbezeichnung ermöglicht.

1 Im Vordergrund steht in dieser Arbeit das Ermitteln von Features und die Überprüfung des Auftretens dieser Features im Text. Die hier aufgelisteten Anwendungsmöglichkeiten können zur Durchführung dieser Aufgaben verwendet werden. Da jedoch Suffix Arrays und nicht Suffix Trees betrachtet werden, wird an einer späteren Stelle der vorliegenden Arbeit, siehe Kapitel 3.1.4.5, die genaue Anwendung, dann bezogen auf Suffix Arrays, erläutert.

- **Suche nach Übereinstimmungen, die Fehler enthalten dürfen**
  Bei der Suche nach bestimmten Zeichenketten in einem Text oder auch dem Vergleich von Zeichenketten miteinander kann es Sinn machen, nicht nur nach exakter Übereinstimmung zu suchen, sondern auch nach einem hohen Grad an Ähnlichkeit. Ähnlichkeit bedeutet so viel wie eine Übereinstimmung, bei der kleinere Fehler erlaubt sind. Eine Lösungsmoglichkeit für dieses Problem bietet das so genannte „k-mismatch problem“[1]. Man legt vorher eine kleine Anzahl k von erlaubten Fehlern fest und versucht, alle Zeichenketten mit maximal k Fehlern innerhalb des Textes zu finden.

3.1.3.4 Suffix Tree für mehrere Texte

Verdeutlicht man sich die in dieser Arbeit gegebene Problemstellung noch einmal, also die Überprüfung, ob mit Features, die auf Text-Suffix-Fragmenten basieren, eine verbesserte Qualität im Hinblick auf die Klassifizierung und das Clustern von natürlichsprachlichen Texten erreicht werden kann, so wird klar, dass das Betrachten nur *eines* natürlichsprachlichen Textes nicht ausreicht. Vielmehr müssen *mehrere* natürlichsprachliche Texte betrachtet und damit auch aufbereitet werden. Aus diesem Grund reicht es nicht aus, den Aufbau eines Suffix Trees zu erläutern, sondern es muss stattdessen der Aufbau eines so genannten „generalisierten“[2] Suffix Trees erläutert werden.
Prinzipiell gelten für diesen Suffix Tree für mehrere Texte die gleichen Bedingungen, die in Definition 8 auf S. 94 der vorliegenden Arbeit genannt wurden. Geändert werden muss die Knotenbezeichnung, da es jetzt nicht mehr ausreichend ist, nur die Position des Suffix im Text anzugeben, sondern vielmehr auch die Information, aus welchem Text das Suffix stammt. Das wird durch einen doppelten Index im Blattknoten erreicht, wobei die erste Position die Information darüber ist, aus welchem Text das Suffix stammt, und die zweite Position, um welches Suffix es sich handelt. Beide Positionen werden in dieser Arbeit durch ein Komma voneinander getrennt. Zusätzlich kann die Bezeichnung des Blattknotens auch mehrere Indizes enthalten, nämlich dann, wenn ein Suffix in mehreren Texten vorkommt. Der auf S. 95 vorgestellte naive Algorithmus kann ebenfalls zum Aufbau eines generalisierten Suffix Trees verwendet werden, nur würde dann nach dem Ende der Verarbeitung des ers-

1 Gusfield (1999), S. 200.
2 Vgl. Gusfield (1999), S. 116.

ten Textes die Schleife ab der Zeile 4 des Algorithmus noch einmal für den kompletten zweiten Text ausgeführt.[1] Das Ergebnis der Durchführung des Algorithmus für die Texte „sitzplatz$“ und „stehplatz$“ sieht man in Abbildung 3.2.

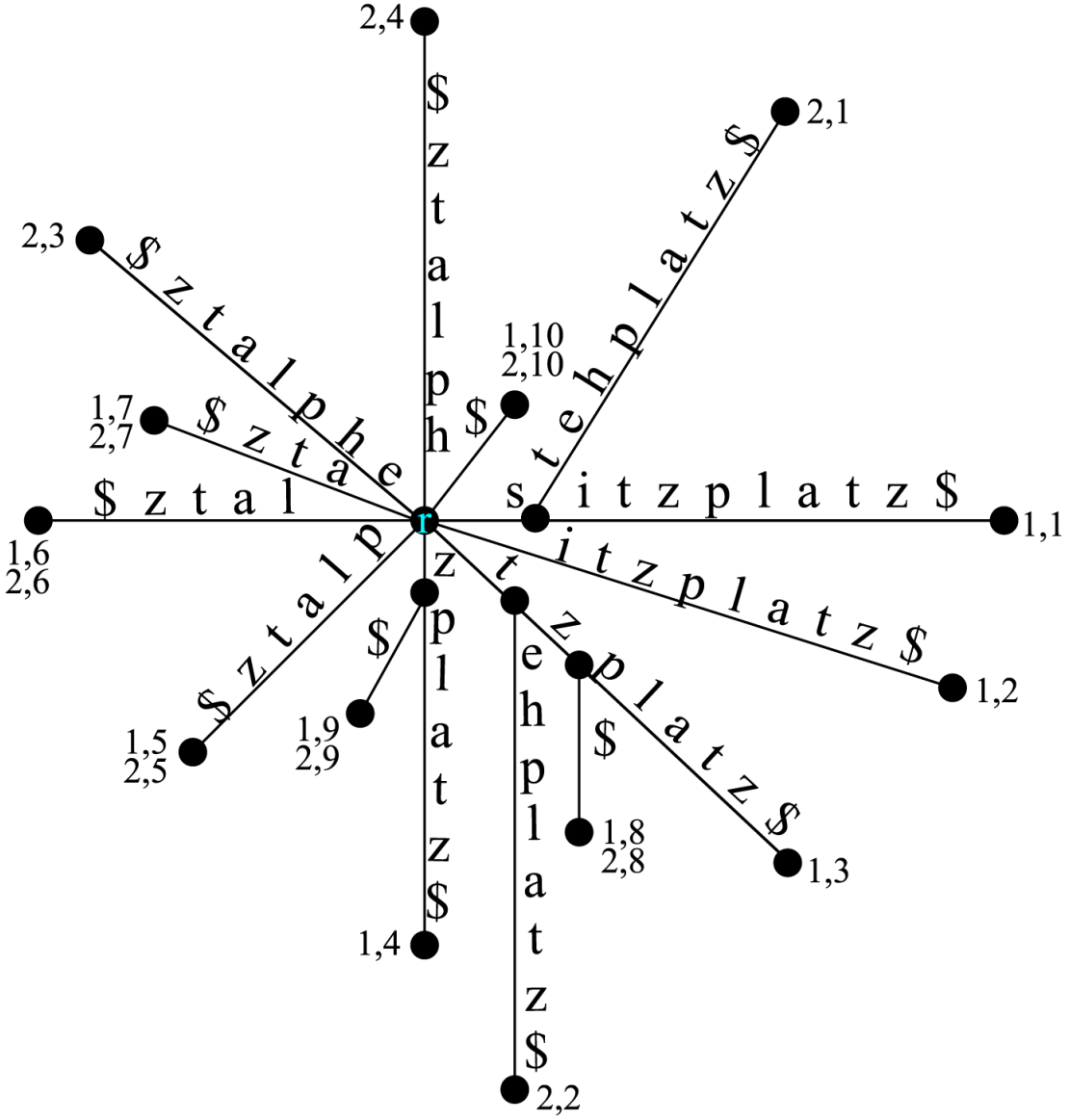


Abbildung 3.2: Darstellung eines generalisierten Suffix Trees

1 Auch hier existieren verbesserte Algorithmen, jedoch stehen diese nicht im Fokus der vorliegenden Arbeit. Prinzipiell werden die verbesserten Algorithmen für die Erstellung des Suffix Trees für einen Text auf den generalisierten Suffix Tree übertragen.

Kann man den generalisierten Suffix Tree mehrerer Texte aufbauen, so macht noch eine andere Anwendungsmöglichkeit für Suffix Trees Sinn:

- **Suchen nach der längsten gemeinsamen Teilzeichenkette zweier oder mehrerer Texte**
  Zunächst wird der generalisierte Suffix Tree für die Texte erstellt. Beim Erstellen erhalten auch die inneren Knoten Informationen darüber, zu welchem Text ihre Pfade gehören. Das bedeutet, sie können entweder nur zu einem Text gehören oder aber zu mehreren Texten. Um die längste gemeinsame Teilzeichenkette zu finden, muss der tiefste innere Knoten oder Blattknoten[1], der alle Textbezeichner enthält, gesucht werden. Der Pfad bis dorthin ist dann die gewünschte Teilzeichenkette. In Abbildung 3.2 ist der tiefste Knoten der inneren Knoten und Blattknoten, die Kantenbeschriftungen aus beiden Texten haben, der Knoten (1,5),(2,5), der das Teilwort „platz$“ abschließt.

Aufgrund der Speichereffizienz[2] werden in dieser Arbeit keine Suffix Trees zur Aufbereitung der natürlichsprachlichen Texte verwendet, sondern Suffix Arrays. Auf diese wird im folgenden Unterkapitel eingegangen.

3.1.4 Suffix Array

3.1.4.1 Definition eines Suffix Arrays

Um die Nachteile des Suffix Trees zu umgehen, entwickelten Manber u.a. (1990) und unabhängig davon Gonnet u.a. (1992) eine neue Datenstruktur: das Suffix Array[3]. Es ist folgerndermaßen definiert[4]:

Definition 9. *Das Suffix Array einer Zeichenkette ist eine Liste aller Suffixe der Zeichenkette in lexikografischer Sortierung.*

1 Die Tiefe dieser Knoten bezieht sich in diesem Fall auf die längste Zeichenkette auf der Kante, die zu diesem Knoten führt.

2 Ein Suffix Array benötigt für den gleichen Text viel weniger Speicherplatz als ein Suffix Tree, vgl. bspw. Navarro u.a. (2000), S. 206, 226; Sadakane (1998), S. 129. Zudem konnte experimentell gezeigt werden, dass trotz einer theoretisch schlechteren Laufzeit der Suchalgorithmen für Suffix Arrays im Vergleich zu den Algorithmen für Suffix Trees die Laufzeit im Experiment bei Suffix Arrays besser oder nahezu gleich ist, vgl. Navarro u.a. (2000), S. 206, 226 f. In Abouelhoda u.a. (2004) zeigen die Autoren, dass jeder Algorithmus, der auf der Datenstruktur Suffix Tree basiert, durch einen auf Suffix Arrays basierenden Algorithmus ersetzt werden kann, der die gleiche Laufzeit wie der Algorithmus für den Suffix Tree benötigt. Des Weiteren ist es schwierig, Suffix Trees effizient im Hinblick auf die Laufzeit zu implementieren, vgl. Schürmann u.a. (2007), S. 309.

3 Bei Gonnet u.a. (1992) heißt es stattdessen PAT array.

4 Vgl. bspw. Ko u.a. (2003), S. 200; Manber u.a. (1990), S. 320; Kim u.a. (2003), S. 186; Kärkkäinen u.a. (2003), S. 943.

Anders als in einem Suffix Tree wird der Text nicht in einer Baumstruktur abgebildet, sondern stattdessen in einer sortierten Liste. Die Sortierung erfolgt dabei lexikografisch vom kleinsten Zeichen bis zum größten Zeichen. Das bedeutet gleichzeitig, dass sich gleiche Präfixe mehrerer Suffixe eines Textes nicht mehr in einem gemeinsamen Pfad im Baum befinden, sondern stattdessen nebeneinander in einer solchen Liste stehen. Nur wenn die Suffixe genau übereinstimmen würden, würden sie an der gleichen Stelle in dieser Liste stehen. Dieser Fall kann jedoch bei der Aufbereitung *eines* Textes durch ein Suffix Array nicht eintreten.[1] Er wird jedoch bei der Betrachtung eines Suffix Arrays für *mehrere* Texte interessant.[2]
Wichtigster Grund für die Verwendung von Suffix Arrays anstelle von Suffix Trees in der vorliegenden Arbeit ist die bessere Speichereffizienz in der Implementierung der Datenstruktur für natürlichsprachliche Texte, denen ein großes Alphabet zu Grunde liegt, und dass der Text, für den das Array erzeugt wird, sich nicht ändert, sondern gleich bleibt.[3] Ein Grund für die Reduzierung des benötigten Speichers liegt darin, dass lediglich die Positionen der Suffixe unter Berücksichtigung der lexikografischen Reihenfolge gespeichert werden und nicht die kompletten Suffixe.[4] Daher wird die oben stehende Definition eines Suffix Arrays leicht abgewandelt[5]:

Definition 10. *Das Suffix Array einer Zeichenkette ist eine Liste aller Anfangspositionen der Suffixe der Zeichenkette, wobei sich die Reihenfolge der Anfangspositionen aus der lexikografisch aufsteigenden Sortierung der dazugehörenden Suffixe der Zeichenkette ergibt.*

Das Suffix Array für das Beispiel aus dem vorangegangenen Kapitel für den Text „sitzplatz“ sieht wie weiter unten stehend aus.[6] Zur Veranschaulichung werden die Suffixe des Textes im Beispiel noch unterhalb der doppelten Linie aufgeführt. In der eigentlichen Implementierung wären sie nicht mehr explizit vorhanden, sondern nur noch die Anfangspositionen der Suffixe im Text. Die kleiner gedruckten Zahlen in der ersten Zeile stellen die Nummerierung der Positionen des Suffix Arrays dar.

1 Vgl. Gonnet u.a. (1992), S. 67 f.
2 Siehe Kapitel 3.1.5, S. 184.
3 Vgl. Gusfield (1999), S. 149.
4 Vgl. Gusfield (1999), S. 149; Ko u.a. (2003), S. 202; Gonnet u.a. (1992), S. 68.
5 Vgl. Sadakane (1998), S. 130; Schürmann u.a. (2007), S. 311. In der zitierten Literatur werden anstatt der Bezeichnung *Anfangspositionen* die Bezeichnungen *Index* oder *Suffixnummer* verwendet.
6 Ein Array, wie es in einer Implementierung als Datenstruktur „aussehen“ würde, wird in der vorliegenden Arbeit in Tabellenform dargestellt. Es handelt sich jedoch nicht um eine Tabelle. Daher werden die Zeilen nicht durchgehend beschriftet. Die kleiner gedruckten Zahlen in der ersten Zeile sind immer die Positionen oder, besser gesagt, der Index des Arrays selbst.

Eine Besonderheit ist auffällig: das Endezeichen „$“ wird als kleinstes Zeichen des Alphabets behandelt.[1]

Ergänze den Text T um das Endezeichen „$“.

T = „sitzplatz$“

Nummeriere alle Zeichen des Textes T von 1 bis $|T|$ durch.

1	2	3	4	5	6	7	8	9	10
s	i	t	z	p	l	a	t	z	$

Füge die Nummern der Suffixe des Textes T in lexikografischer Reihenfolge in ein Array ein.

1	2	3	4	5	6	7	8	9	10
10	7	2	6	5	1	8	3	9	4
$	a t z $	i t z p l a t z $	l a t z $	p l a t z $	s i t z p l a t z $	t z $	t z p l a t z $	z $	z p l a t z $

3.1.4.2 Überführung eines Suffix Trees in ein Suffix Array

Die einfachste Möglichkeit, ein Suffix Array für einen Text T zu erstellen, ist die Überführung eines Suffix Trees für T in ein Suffix Array. Dabei würde man zunächst den Suffix Tree für T erzeugen und ihn dann in lexikografischer Reihenfolge traversieren. Das bedeutet, man würde jede Kante des Baumes lexikografisch aufsteigend bis zu allen Blattknoten dieser Kante „ablaufen“ und jeweils die Beschriftung oder vielmehr die Bezeichnung des Blattknotens, da ja nur die Suffixpositionen im Array gespeichert werden, als Suffix in das Suffix Array schreiben.[2] Diese Traversierung des Suffix Trees und der gleichzeitige Aufbau des Suffix Arrays können zwar in li-

1 Vgl. Gusfield (1999), S. 149; Ko u.a. (2003), S. 202; Sadakane (1998), S. 130. Das ist auch der Fall, wenn nicht das „$“ als eindeutiges Endezeichen verwendet wird, vgl. bspw. Kim u.a. (2003), S. 187; Gonnet u.a. (1992), S. 68. Der umgekehrte Fall, also dass das Endezeichen als das größte Zeichen des Alphabets behandelt wird, existiert ebenfalls, vgl. Abouelhoda u.a. (2004), S. 56.

2 Vgl. Gusfield (1999), S. 150 f.

nearer Zeit erfolgen, jedoch lässt sich durch diese Methode die Speichereffizienz des Suffix Arrays nicht vollständig ausnutzen, da zunächst der Suffix Tree und damit die speicherintensive Form der Datenstruktur erzeugt werden müsste.[1] Aus diesem Grund existieren mehrere Algorithmen, die direkt aus einem Text T ein Suffix Array dieses Textes in linearer Zeit und mit linearem Speicherverbrauch erzeugen.[2]

3.1.4.3 Algorithmus zur direkten Erstellung eines Suffix Arrays nach Ko und Aluru für einen Text

3.1.4.3.1 Definition von Begriffen

Mit dem im Folgenden vorgestellten Algorithmus ist es möglich, direkt aus einem Text T in linearer Laufzeit ein entsprechendes Suffix Array zu erstellen, ohne vorher einen Suffix Tree für diesen Text erstellen zu müssen. Die Beschreibung des Algorithmus ist Ko u.a. (2003) und Ko u.a. (2005) entnommen.
Der Text, für den das Suffix Array erstellt werden soll, wird mit

Definition 11. $T = T[1...|T|] = t_1 t_2 \dots t_{|T|}$

definiert. Als letztes Zeichen dieses Textes wird, wie bereits beschrieben, ein Zeichen angehängt, das sonst nirgendwo in diesem Text auftaucht.[3] Das Suffix $T[i]$ ist definiert durch[4]:

Definition 12. $T[i] = T[i...|T|] = t_i t_{i+1} \dots t_{|T|}$.

Soll dieses Suffix im Array gespeichert werden, so wird nur seine Position, also i, gespeichert[5].
Eine weitere Definition ist noch wichtig[6]:

Definition 13. *Vergleicht man zwei Zeichenketten, in diesem Fall Suffixe, α und β lexikografisch miteinander, so soll $\alpha \prec \beta$ bedeuten, dass α lexikografisch kleiner als β ist.*

1 Vgl. Ko u.a. (2003), S. 200.
2 Daneben existiert auch ein Algorithmus, der einen Suffix Tree durch ein so genanntes „enhanced Suffix Array“, Abouelhoda u.a. (2004), simuliert und es damit ermöglicht, den Speicherplatzbedarf eines Suffix Trees zu senken. Diese Datenstruktur wird im Rahmen der vorliegenden Arbeit nicht weiter betrachtet.
3 In der Literatur wird das „$“ verwendet, vgl. Ko u.a. (2003), S. 202; Ko u.a. (2005), S. 145. Es sind auch andere Zeichen möglich, vgl. Fußnote 1 auf Seite 105.
4 Vgl. Ko u.a. (2003), S. 202; Ko u.a. (2005), S. 145.
5 Vgl. Ko u.a. (2003), S. 202; Ko u.a. (2005), S. 145.
6 Vgl. Ko u.a. (2003), S. 202; Ko u.a. (2005), S. 145.

3.1.4.3.2 Ablauf des Algorithmus

3.1.4.3.2.1 Übersicht über den Gesamtalgorithmus

Der Algorithmus lässt sich in sieben Schritte unterteilen.[1]. Diese sind:

1. Klassifizieren der Suffixe des Textes T nach Typen
2. Sortieren aller Suffixe nach ihrem ersten Zeichen
3. Konstruktion von m Listen
4. Sortieren aller Typ-L- oder Typ-S-Suffixe mit Hilfe der Listen
5. Konstruktion einer neuen Zeichenkette T' aufgrund der zu sortierenden Suffixe
6. Rekursive Anwendung des Algorithmus, um T' zu sortieren
7. Konstruktion des Suffix Arrays aufgrund der Vorsortierung der Typ-L- oder Typ-S-Suffixe

Als Gesamtalgorithmus sieht das wie folgt aus[2]:

Algorithmus 6 erstelleSuffixArray

Eingabe: Text T, für den ein Suffix Array erstellt werden soll

Ausgabe: Suffix Array A des Textes T

1: Typ, $sortS$ = KLASSIFIZIERESUFFIXE(T)
2: A = SORTIERESUFFIXEERSTESZEICHEN(T)
3: $Dist$, m-Listen = KONSTRUIEREMLISTEN(T, $sortS$, A)
4: C = SORTIERESUFFIXEMITLISTEN(A, Typ, $sortS$, T, m-Listen)
5: **if** Array C noch nicht sortiert **then**
6: $\quad T'$ = KONSTRUIEREZEICHENKETTET'(C, $sortS$)
7: $\quad C$ = ERSTELLESUFFIXARRAY(T')
8: **end if**
9: A = ABSCHLIESSENDESORTIERUNGARRAYA(A, C, $sortS$, T, Typ)

1 Vgl. Ko u.a. (2005), S. 151. Diese Schritte werden in den folgenden Kapiteln der vorliegenden Arbeit genauer beschrieben.

2 Die Algorithmusnamen in Kapitälchen bedeuten, dass den links stehenden Variablen das Ergebnis der Ausführung der entsprechenden Algorithmen zugewiesen wird. Die entsprechenden Algorithmen werden in den folgenden Kapiteln erläutert.

3.1.4.3.2.2 Klassifizieren der Suffixe eines Textes T nach Typen

Es gilt Folgendes[1]:

Definition 14. *Ein Suffix $T[i]$ soll von Typ-S sein, wenn gilt: $T[i] \prec T[i+1]$. Gilt der umgekehrte Fall, also $T[i+1] \prec T[i]$, dann handelt es sich bei Suffix $T[i]$ um ein Suffix von Typ-L. Eine Besonderheit ist das letzte Zeichen des Textes, hier "$". Es soll sowohl von Typ-L als auch von Typ-S sein. Die Bezeichnung dafür ist L/S.*[2]

Algorithmus 7 klassifiziereSuffixe

Eingabe: Text T mit Länge $|T|$

Ausgabe: Array Typ mit Typen der Suffixe von T und ein boolean-Wert $sortS$, der angibt, ob die Suffixe vom Typ-S sortiert werden sollen

```
1: sortS = false
2: countS = 0
3: for i = 1, ..., |T| − 1 do
4:     Vergleiche Suffix T[i] mit Suffix T[i + 1]
5:     if T[i] ≺ T[i + 1] then
6:         Typ[i] = S
7:         countS = countS + 1
8:     else
9:         Typ[i] = L
10:    end if
11: end for
12: if countS ≥ |T| − 1 − countS then
13:    sortS = true
14: end if
15: Typ[|T|] = L/S
```

Der Algorithmus, wie man später noch sehen wird, beruht darauf, dass in den meisten Texten entweder weniger Typ-S-Suffixe oder weniger Typ-L-Suffixe vorhanden sind und dass eine Vorsortierung dieser kleinen Menge an Suffixen dafür sorgt, dass die Gesamtzahl an Suffixen in linearer Zeit sortiert werden kann.[3]

1 Vgl. Ko u.a. (2003), S. 202; Ko u.a. (2005), S. 145.

2 Im Algorithmus und den erläuternden Texten wird diese Bezeichnung zunächst so aus der Originalquelle übernommen. In der späteren Implementiertung weicht die Verfasserin davon ab, siehe Kapitel 3.1.4.4.2.2 auf S. 157.

3 Vgl. Ko u.a. (2003), S. 204 f.; Ko u.a. (2005), S. 148.

Durch einen einfachen Durchlauf des kompletten Textes und jeweils die Betrachtung zweier nebeneinanderstehenden Suffixe können alle Suffixe des Textes in Typ-L oder S klassifiziert werden.[1] Die folgenden Beispiele für die Texte T_1 =„sitzplatz$“ und T_2 =„stehplatz$“ zeigen die Einteilung der Suffixe in Typ-S und Typ-L.

	1	2	3	4	5	6	7	8	9	10
T_1	s	i	t	z	p	l	a	t	z	$
Typ	L	S	S	L	L	L	S	S	L	L/S
T_2	s	t	e	h	p	l	a	t	z	$
Typ	S	L	S	S	L	L	S	S	L	L/S

3.1.4.3.2.3 Sortieren aller Suffixe nach ihrem ersten Zeichen

Nachdem die Typen der Suffixe festgelegt sind, erzeugt man eine erste Version des späteren Suffix Arrays, das Array A[2], in dem alle Suffixe des Textes T sortiert nach ihrem ersten Zeichen und nach der Reihenfolge ihres Auftretens im Text in dieses Array gespeichert werden.[3] Zwei Dinge sind bei diesem Vorgang zu berücksichtigen:

1. Das als Abschlusszeichen hinzugefügte “$” ist immer das kleinste Zeichen des Textes.[4]

2. Im Array A werden nur die Positionen der Suffixe gespeichert, nicht das komplette Suffix.[5]

Zusätzlich werden für das so erzeugte Array Buckets[6] gebildet, die die Einteilung zwischen unterschiedlichen Zeichen in diesem Array bilden.[7] Im Algorithmus ist es nötig, auf das erste Zeichen eines Suffixes zuzugreifen. Das wird über einen Zugriff wie in einem zweidimensionalen Array dargestellt. $T[i][j]$ bedeutet also, es wird auf das Zeichen an Position j im Suffix $T[i]$ zugegriffen. Ist bspw. T =“arbeit$“ und es soll auf das Zeichen $T[3][3]$ zugegriffen werden, so ergibt sich „i“.

1 Vgl. Ko u.a. (2003), S. 202; Ko u.a. (2005), S. 145 f.
2 Bei A handelt es sich um einen Variablennamen für das Suffix Array. Er wird von der Verfasserin aus den Quellen übernommen, ist jedoch beliebig wählbar und damit kein feststehendes Symbol. Das gilt ebenso für die Namen der anderen Arrays in der vorliegenden Arbeit.
3 Vgl. Ko u.a. (2003), S. 203; Ko u.a. (2005), S. 146 f.
4 Vgl. Ko u.a. (2003), S. 202; Ko u.a. (2005), S. 145.
5 Vgl. Ko u.a. (2003), S. 202; Ko u.a. (2005), S. 145.
6 Die wörtliche Übersetzung ist „Eimer“ oder „Kübel“. Gemeint sind hier eher „Töpfe“ im Sinne von Klassen, d.h., man sortiert gleiche Anfangszeichen in ein Bucket, hier zunächst das erste Zeichen eines Suffixes, und teilt so das gesamte Array in kleinere „Portionen“ von Suffixen auf, die jeweils mit dem gleichen Zeichen beginnen.
7 Vgl. Ko u.a. (2003), S. 203; Ko u.a. (2005), S. 147.

Algorithmus 8 sortiereSuffixeErstesZeichen

Eingabe: Text T mit Länge $|T|$

Ausgabe: Array A mit den Suffixpositionen des Textes T sortiert nach ihrem ersten Zeichen und entsprechenden Bucketgrenzen

1: Erzeuge Array A mit Länge T
2: **for** $i = 1, ..., |T|$ **do**
3: Betrachte Suffix $T[i]$
4: Füge i in A an der Stelle ein, an die das entsprechende Suffix $T[i]$ lexikografisch aufgrund seines ersten Zeichens $T[i][1]$ gehört, der Index im Array A ist j
5: **if** kein Bucket für dieses erste Zeichen vorhanden **then**
6: Erzeuge Bucketanfang für $T[i][1]$, das ist $bucketanfang_{T[i][1]}$
7: $bucketanfang_{T[i][1]} = j$
8: Erzeuge Bucketende für $T[i][1]$, das ist $bucketende_{T[i][1]}$
9: **end if**
10: $bucketende_{T[i][1]} = j$
11: **end for**

Für die beiden Beispiele sieht das Array A wie folgt aus, wobei doppelte vertikale Linien das Ende eines Buckets markieren. Zur Verdeutlichung stehen die ersten Zeichen nochmal unter den entsprechenden Suffixpositionen:

	1	2	3	4	5	6	7	8	9	10
A_{T_1}	10	7	2	6	5	1	3	8	4	9
	\$	a	i	l	p	s	t	t	z	z

	1	2	3	4	5	6	7	8	9	10
A_{T_2}	10	7	3	4	6	5	1	2	8	9
	\$	a	e	h	l	p	s	t	t	z

3.1.4.3.2.4 Konstruktion von m Listen

Um das Suffix Array direkt sortiert erstellen zu können, wird die kleinere Anzahl an Suffixen, also entweder Typ-L oder Typ-S, vorsortiert.[1] Um das in linearer Zeit leisten zu können, werden m Listen mit Buckets erstellt, basierend auf der L- oder

1 Vgl. Ko u.a. (2003), S. 205; Ko u.a. (2005), S. 148.

S-Distanz der Suffixe des Textes.[1] Mit Hilfe dieser Listen können dann die vorzusortierenden Suffixe sortiert werden.

Zunächst sind weitere Definitionen wichtig[2]:

Definition 15. *Die Teilzeichenkette $t_i \dots t_j$ wird als Typ-S-Teilzeichenkette bezeichnet, wenn sowohl das Suffix an Position i des Textes T als auch das Suffix an Position j des Textes T beides Typ-S-Suffixe sind und im Text T dazwischen nur Typ-L-Suffixe liegen. Analog gilt die Definition ebenfalls für eine Typ-L-Teilzeichenkette.*

Definition 16. *Die S-Distanz eines jeden Suffix $T[i..|T|]$ ist die Distanz von seiner Startposition i im Text T zur nächsten Position eines Typ-S-Suffix links von der Position des Suffix $T[i..|T|]$ im Text T. Die Position i selbst wird dabei ausgeschlossen. Sollte links von der Position des Suffix $T[i..|T|]$ im Text T kein Typ-S-Suffix liegen, so ist die S-Distanz 0.*

Definition 17. *Die L-Distanz eines jeden Suffix $T[i..|T|]$ ist die Distanz von seiner Startposition i im Text T zur nächsten Position eines Typ-L-Suffix links von der Position des Suffix $T[i..|T|]$ im Text T. Die Position i selbst wird dabei ausgeschlossen. Sollte links von der Position des Suffix $T[i..|T|]$ im Text T kein Typ-L-Suffix liegen, so ist die L-Distanz 0.*

Um die m Listen zu konstruieren, muss zunächst die S- bzw. L-Distanz eines jeden Suffixes bestimmt werden. Welche der Distanzen bestimmt wird, hängt davon ab, welche Suffixe vorsortiert werden sollen: die S-Distanz, wenn Typ-S-Suffixe sortiert werden sollen, und die L-Distanz, wenn Typ-L-Suffixe sortiert werden sollen. Um diese Distanz zu bestimmen, wird T von links nach rechts durchlaufen und man merkt sich dabei jeweils die Distanz von der aktuellen Position zur letzten Position eines Typ-S- bzw. Typ-L-Suffix. Diese Distanz wird in einem zusätzlichen Array *Dist* gespeichert, wobei die Distanz von Suffix $T[i..|T|]$ in $Dist[i]$ gespeichert wird.[3]

1 Vgl. Ko u.a. (2003), S. 205; Ko u.a. (2005), S. 148 f. Der Parameter m bezeichnet die größte aufgetretene Distanz.

2 Vgl. Ko u.a. (2003), S. 205, 207; Ko u.a. (2005), S. 148, 150 f.

3 Die Position des letzten Elements in einem Array A wird mit *A.length* angegeben. Auch bei den folgenden Algorithmen besagt ein Algorithmusname in Kapitälchen, dass der genannte Algorithmus oder die genannte Funktion aufgerufen und damit ausgeführt wird. Wird ein Algorithmus oder eine Funktion nicht nur aufgerufen, sondern steht sein oder ihr Name rechts von einem Zuweisungsoperator, so bedeutet das, dass die vom Algorithmus oder von der Funktion zurückgegebenen Werte den Variablen links vom Zuweisungsoperator zugewiesen werden.

Algorithmus 9 konstruiereMListen

Eingabe: Text T mit Länge $|T|$, $sortS$, Array A_T

Ausgabe: Array $Dist$ mit den L- oder S-Distanzen der Suffixe des Textes T und m Listen

```
1: Dist, m = BERECHNEDISTANZ(T, sortS)
2: Erzeuge m Listen
3: for i = 1, ..., A_T.length do
4:     Lies A_T[i] aus, das ist x
5:     Lies Dist[x] aus, das ist j
6:     if j > 0 then
7:         Speichere x an der nächsten Position in Liste j ab
8:     end if
9: end for
```

Algorithmus 10 berechneDistanz

```
1: function BERECHNEDISTANZ(T,sortS)
2:     m = 0
3:     Erzeuge Array Dist mit der Länge |T|
4:     Dist[1] = 0
5:     distanz = −1
6:     if sortS == true then
7:         if Typ[1] == S then
8:             distanz = 1
9:         else
10:            distanz = 0
11:        end if
12:    else
13:        if Typ[1] == L then
14:            distanz = 1
15:        else
16:            distanz = 0
17:        end if
18:    end if
19:    for i = 2, ..., |T| do
20:        Dist[i] = distanz
21:        if sortS == true then
22:            if Typ[i] == S then
```

Algorithmus 10 berechneDistanz (Teil 2)

```
23:             if distanz > m then
24:                 m = distanz
25:             end if
26:             distanz = 1
27:         else
28:             distanz = distanz + 1
29:         end if
30:     else
31:         if Typ[i] == L then
32:             if distanz > m then
33:                 m = distanz
34:             end if
35:             distanz = 1
36:         else
37:             distanz = distanz + 1
38:         end if
39:     end if
40: end for
41: Gib Dist und m zurück
42: end function
```

Für die beiden Beispiele[1] ergeben sich folgende Distanzen und folgende Arrays *Dist*:

	1	2	3	4	5	6	7	8	9	10
T_1	s	i	t	z	p	l	a	t	z	$
Typ	L	S	S	L	L	L	S	S	L	L/S
T_2	s	t	e	h	p	l	a	t	z	$
Typ	S	L	S	S	L	L	S	S	L	L/S

	1	2	3	4	5	6	7	8	9	10
$Dist_{T_1}$	0	0	1	1	2	3	4	1	1	2

	1	2	3	4	5	6	7	8	9	10
$Dist_{T_2}$	0	0	1	2	3	1	1	2	3	1

1 Für das erste Beispiel T_1 werden die Typ-S-Suffixe vorsortiert, da hier 5 Typ-S-Suffixe existieren gegenüber 6 Typ-L-Suffixen, und für das zweite Beispiel T_2 die Typ-L-Suffixe, da hier 5 Typ-L-Suffixe vorhanden sind gegenüber 6 Typ-S-Suffixen.

Die jeweiligen Distanzen lassen sich also durch einen Durchlauf des Textes erzeugen, d.h., in linearer Zeit.[1]
Zusätzlich merkt man sich die größte auftretende Distanz m. Dabei handelt es sich um die Anzahl der Listen, die benötigt werden, um die Typ-L- oder S-Suffixe zu sortieren. Die Listen werden so erstellt, dass Liste j ($1 \leq j \leq m$) die Suffixpositionen enthält, die eine L- oder S-Distanz von j aufweisen.[2] Diese Suffixpositionen werden in der entsprechenden Liste in der Reihenfolge ihres Auftretens in Array A gespeichert. Man durchläuft also A von links nach rechts und greift auf $Dist[A[i]]$ zu, um die richtige Liste für Suffix $T[A[i]]$ zu finden.
Für das Erzeugen der Listen zum Vorsortieren der Typ-S-Suffixe aus T_1 werden folgende Schritte durchgeführt[3]:

	1	2	3	4	5	6	7	8	9	10
A_{T_1}	10	7	2	6	5	1	3	8	4	9
$Dist_{T_1}$	0	0	1	1	2	3	4	1	1	2

$i = 1$
$x = A_{T_1}[i] = A_{T_1}[1] = 10$
$j = Dist_{T_1}[x] = Dist_{T_1}[10] = 2$
$\Rightarrow$ Textposition 10 in Liste 2 an Listenposition 0 speichern
$i = 2$
$x = A_{T_1}[i] = A_{T_1}[2] = 7$
$j = Dist_{T_1}[x] = Dist_{T_1}[7] = 4$
$\Rightarrow$ Textposition 7 in Liste 4 an Listenposition 0 speichern
$i = 3$
$x = A_{T_1}[i] = A_{T_1}[3] = 2$
$j = Dist_{T_1}[x] = Dist_{T_1}[2] = 0$
$\Rightarrow$ Textposition 2 wird nicht gespeichert, da $j > 0$ sein muss
...
$i = 10$
$x = A_{T_1}[i] = A_{T_1}[10] = 9$
$j = Dist_{T_1}[x] = Dist_{T_1}[9] = 1$
$\Rightarrow$ Textposition 9 in Liste 1 an Listenposition 3 speichern

1 Vgl. Ko u.a. (2003), S. 205; Ko u.a. (2005), S. 148.
2 Vgl. Ko u.a. (2003), S. 205; Ko u.a. (2005), S. 148 f.
3 Der Ablauf des Algorithmus 9 ab Zeile 3 wird nach den Arrays dargestellt. Die Werte werden den Beispielarrays entnommen.

Die Listen für den Text T_1 sehen wie folgt aus[1]:

	0	1	2	3
1:	3	8	4	9
2:	10	5		
3:	6			
4:	7			

Für das Erzeugen der Listen zum Vorsortieren der Typ-L-Suffixe für den Text T_2 werden folgende Schritte durchgeführt:

	1	2	3	4	5	6	7	8	9	10
A_{T_2}	10	7	3	4	6	5	1	2	8	9
$Dist_{T_2}$	0	0	1	2	3	1	1	2	3	1

$i = 1$
$x = A_{T_2}[i] = A_{T_2}[1] = 10$
$j = Dist_{T_2}[x] = Dist_{T_2}[10] = 1$
$\Rightarrow$ Textposition 10 in Liste 1 an Listenposition 0 speichern
$i = 2$
$x = A_{T_2}[i] = A_{T_2}[2] = 7$
$j = Dist_{T_2}[x] = Dist_{T_2}[7] = 1$
$\Rightarrow$ Textposition 7 in Liste 1 an Listenposition 1 speichern
$i = 3$
$x = A_{T_2}[i] = A_{T_2}[3] = 3$
$j = Dist_{T_2}[x] = Dist_{T_2}[3] = 1$
$\Rightarrow$ Textposition 3 in Liste 1 an Listenposition 2 speichern
...
$i = 10$
$x = A_{T_2}[i] = A_{T_2}[10] = 9$
$j = Dist_{T_2}[x] = Dist_{T_2}[9] = 3$
$\Rightarrow$ Textposition 9 in Liste 3 an Listenposition 1 speichern

Die Listen für den Text $T2$ sehen wie folgt aus:

	0	1	2	3
1:	10	7	3	6
2:	4	8		
3:	5	9		

1 Die Listen sind horizontal dargestellt. Die erste Zeile entspricht den Indizes der Listen.

3.1.4.3.2.5 Sortieren aller Typ-L- oder Typ-S-Suffixe mit Hilfe der Listen

Algorithmus 11 sortiereSuffixeMitListen

Eingabe: Array A, Array Typ, $sortS$, Text T, m Listen

Ausgabe: Array C mit den sortierten Typ-L- oder S-Suffixen

1: C = ERZEUGEARRAYC(A, Typ, $sortS$, T)
2: m Listen = ERZEUGELISTENBUCKETS(m Listen)
3: R = ERZEUGEARRAYR(C)
4: $lptr$ = ERZEUGEARRAYLPTR(C)
5: C = SORTIERESUFFIXEMITRUNDLPTR(R, $lptr$, $sortS$)

Das Sortieren der kleineren Anzahl der Suffixe erfolgt durch wiederholtes Erzeugen von Buckets und das Durchlaufen der erzeugten m Listen.[1] Dabei wird die Sortierung schrittweise durchgeführt. Betrachtet man die Liste 1, so werden die Suffixe nach ihren ersten beiden Zeichen sortiert, betrachtet man Liste 2, so werden die Suffixe nach ihren ersten drei Zeichen sortiert usw.
Der erste Schritt ist das Erzeugen eines Arrays C, das entweder alle Typ-S-Suffixe oder alle Typ-L-Suffixe enthält.[2] Das Erstellen erfolgt durch das Kopieren der entsprechenden Suffixpositionen aus Array A. Als Algorithmus sieht das wie folgt aus:

Algorithmus 12 erzeugeArrayC

Eingabe: Array A, Array Typ, $sortS$, Text T

Ausgabe: Array C mit Buckets aufgrund des ersten Zeichens der Suffixe

1: Erzeuge Array C
2: $j = 1$
3: **for** $i = 1, ..., A.length$ **do**
4: **if** $sortS == true$ **then**
5: Lies $A[i]$ aus, das ist x
6: **if** $Typ[x] == S$ **then**
7: **if** $j > 1$ **then**
8: **if** $T[x][1] \neq T[C[j-1]][1]$ **then**[3]
9: Erzeuge ein neues Bucket in C
10: **end if**
11: **end if**

1 Vgl. Ko u.a. (2003), S. 205; Ko u.a. (2005), S. 149.
2 Vgl. Ko u.a. (2005), S. 151 f.
3 Das erste Zeichen des aktuell betrachteten Suffixes ist ungleich dem ersten Zeichen des Suffixes in C an der vorherigen Position.

Algorithmus 12 erzeugeArrayC (Teil 2)

```
12:             Speichere x in C[j]
13:             j = j + 1
14:         end if
15:     else
16:         Lies A[i] aus, das ist x
17:         if Typ[i] == L then
18:             if j > 1 then
19:                 if T[x][1] ≠ T[C[j − 1]][1] then
20:                     Erzeuge ein neues Bucket in C
21:                 end if
22:             end if
23:             Speichere x in C[j]
24:             j = j + 1
25:         end if
26:     end if
27: end for
```

Für die beiden Beispiele ergeben sich folgende Arrays C:

	1	2	3	4	5
C_{T_1}	10	7	2	3	8

	1	2	3	4	5
C_{T_2}	10	6	5	2	9

Für beide Arrays wurden Buckets aufgrund des ersten Zeichens der enthaltenen Suffixe erzeugt.[1] Wie man daran bereits erkennen kann, sind die zu sortierenden Typ-L-Suffixe schon sortiert, da sie sich alle in einem eigenen Bucket befinden. Im Fall der Typ-S-Suffixe ist das hintere Bucket, also die Positionen 4 und 5 aus Array C_{T_1}, noch nicht sortiert, da sich hier zwei Suffixe in einem Bucket befinden.
Das Prinzip der Sortierung wäre es nun, die erstellten Listen zu durchlaufen und das jeweils betrachtete Suffix in Array C an den Anfang seines Buckets, für Typ-S-Suffixe, oder an das Ende seines Buckets, für Typ-L-Suffixe, zu tauschen.[2] Um das eigentliche Vertauschen nicht durchführen zu müssen, werden für die Listen ebenfalls Buckets erstellt und zwei Zusatzarrays erzeugt, mit deren Hilfe dann die Sortierung

1 Das ist, wie bereits gesagt, erkennbar an den doppelten vertikalen Linien.
2 Vgl. Ko u.a. (2003), S. 205 f.; Ko u.a. (2005), S. 149.

vorgenommen wird.[1] Der Algorithmus für die Erzeugung der Listenbuckets ist der folgende[2]:

Algorithmus 13 erzeugeListenbuckets

Eingabe: m Listen, die zuvor in Algorithmus 9 erzeugt wurden

Ausgabe: m Listen mit Buckets aufgrund des Anfangszeichens der enthaltenen Suffixe

```
1: for j = 1, ..., m do
2:     Lies Liste j aus, das ist y
3:     for i = 1, ..., y.length do
4:         if i > 1 then
5:             if T[y[i]] ≠ T[y[i − 1]] then
6:                 Erzeuge eine neue Bucketgrenze, so dass Position i
                   Anfangsposition des neuen Buckets ist und Position i − 1
                   Endposition des vorher stehenden Buckets ist
7:             end if
8:         end if
9:     end for
10: end for
```

Die Listen mit Buckets sehen für den Text T_1 wie folgt aus:

1:	3	8	4	9
2:	10	5		
3:	6			
4:	7			

Jetzt erfolgt die Erstellung eines Arrays R[3], wobei gilt: wenn $C[i] = j$, dann ist $R[j] = l$, wobei l die Endposition für Typ-S-Suffixe oder die Anfangsposition für Typ-L-Suffixe des Buckets ist, das j enthält. D.h., zur Erstellung wird C von rechts nach links für Typ-S oder von links nach rechts für Typ-L durchlaufen und die Informationen werden in R gespeichert. Alle anderen Positionen des Arrays R erhalten eine -1 als Inhalt.

1 Vgl. Ko u.a. (2005), S. 151 f.

2 Um auf eine Position in einer Liste zuzugreifen, wird in der vorliegenden Arbeit der Zugriff wie bei einem Array dargestellt. So bedeutet bspw. $y[5]$, dass in Liste y auf Position 5 zugegriffen wird.

3 Vgl. Ko u.a. (2005), S. 152.

Algorithmus 14 erzeugeArrayR

Eingabe: Array C, $sortS$

Ausgabe: Array R

```
1: Erzeuge Array R und initialisiere es mit −1
2: if sortS == true then
3:     for i = C.length, ..., 1 do
4:         Lies C[i] aus, das ist x
5:         Speichere die Endposition des Buckets von C[i] in R[x]
6:     end for
7: else
8:     for i = 1, ..., C.length do
9:         Lies C[i] aus, das ist x
10:        Speichere die Anfangsposition des Buckets von C[i] in R[x]
11:    end for
12: end if
```

Der Ablauf des Algorithmus, um Array R für Text T_1 zu erstellen, ist der folgende:

$i = 5$

$x = C_{T_1}[i] = C_{T_1}[5] = 8$

$\Rightarrow R_{T_1}[8] =$ Endposition des Buckets in C_{T_1} von Suffix $T_1[8] = 5$

$i = 4$

$x = C_{T_1}[i] = C_{T_1}[4] = 3$

$\Rightarrow R_{T_1}[3] =$ Endposition des Buckets in C_{T_1} von Suffix $T_1[3] = 5$

da sich die Suffixe $T_1[3]$ und $T_1[8]$ im gleichen Bucket in C_{T_1} befinden

$i = 3$

$x = C_{T_1}[i] = C_{T_1}[3] = 2$

$\Rightarrow R_{T_1}[2] =$ Endposition des Buckets in C_{T_1} von Suffix $T_1[2] = 3$

$i = 2$

$x = C_{T_1}[i] = C_{T_1}[2] = 7$

$\Rightarrow R_{T_1}[7] =$ Endposition des Buckets in C_{T_1} von Suffix $T_1[7] = 2$

$i = 1$

$x = C_{T_1}[i] = C_{T_1}[1] = 10$

$\Rightarrow R_{T_1}[10] =$ Endposition des Buckets in C_{T_1} von Suffix $T_1[10] = 1$

Für den Text T_1 ergibt sich folgendes Array R:

	1	2	3	4	5	6	7	8	9	10
R_{T_1}	-1	3	5	-1	-1	-1	2	5	-1	1

Der Ablauf des Algorithmus, um Array R für Text T_2 zu erstellen, ist der folgende:

$i = 1$
$x = C_{T_2}[i] = C_{T_2}[1] = 10$
$\Rightarrow R_{T_2}[10] =$ Anfangsposition des Buckets in C_{T_2} von Suffix$T_2[10] = 1$

$i = 2$
$x = C_{T_2}[i] = C_{T_2}[2] = 6$
$\Rightarrow R_{T_2}[6] =$ Anfangsposition des Buckets in C_{T_2} von Suffix $T_2[6] = 2$

$i = 3$
$x = C_{T_2}[i] = C_{T_2}[3] = 5$
$\Rightarrow R_{T_2}[5] =$ Anfangsposition des Buckets in C_{T_2} von Suffix $T_2[5] = 3$

$i = 4$
$x = C_{T_2}[i] = C_{T_2}[4] = 2$
$\Rightarrow R_{T_2}[2] =$ Anfangsposition des Buckets in C_{T_2} von Suffix $T_2[4] = 4$

$i = 5$
$x = C_{T_2}[i] = C_{T_2}[5] = 9$
$\Rightarrow R_{T_2}[9] =$ Anfangsposition des Buckets in C_{T_2} von Suffix $T_2[5] = 5$

Für Text T_2 entsteht folgendes Array R:

	1	2	3	4	5	6	7	8	9	10
R_{T_2}	-1	4	-1	-1	3	2	-1	-1	5	1

Das zweite Hilfsarray ist das Array *lptr*, wobei gilt: wenn i für Typ-S die letzte für Typ-L die erste Position eines Buckets in C ist, dann ist $lptr[i] = j$, wobei j für Typ-S der aktuelle Anfang oder für Typ-L das aktuelle Ende dieses Buckets ist.[1] Alle anderen Positionen erhalten eine -1.

Algorithmus 15 erzeugeArrayLptr
Eingabe: Array C, $sortS$
Ausgabe: Array $lptr$
1: Erzeuge Array $lptr$ und initialisiere es mit -1
2: **if** $sortS == true$ **then**
3: **for** $i = C.length, ..., 1$ **do**

1 Vgl. Ko u.a. (2005), S. 152.

Algorithmus 15 erzeugeArrayLptr (Teil 2)

```
4:        if i ist Endposition eines Buckets then
5:            merker = i
6:        end if
7:        if i ist Anfangsposition eines Buckets then
8:            lptr[merker] = i
9:        end if
10:   end for
11: else
12:   for i = 1, ..., C.length do
13:       if i ist Anfangsposition eines Buckets then
14:           merker = i
15:       end if
16:       if i ist Endposition eines Buckets then
17:           lptr[merker] = i
18:       end if
19:   end for
20: end if
```

Der Ablauf des Algorithmus, um Array *lptr* für Text T_1 zu erstellen ist der folgende:
$i = 5$
Es handelt sich um eine Endposition eines Buckets in C_{T_1}.
$\Rightarrow merker = i = 5$
$i = 4$
Es handelt sich um eine Anfangsposition eines Buckets in C_{T_1}.
$\Rightarrow lptr[merker] = i = lptr[5] = 4$
$i = 3$
Es handelt sich um eine Endposition eines Buckets in C_{T_1}.
$\Rightarrow merker = i = 3$
Es handelt sich um eine Anfangsposition eines Buckets in C_{T_1}.
$\Rightarrow lptr[merker] = i = lptr[3] = 3$
$i = 2$
Es handelt sich um eine Endposition eines Buckets in C_{T_1}.
$\Rightarrow merker = i = 2$
Es handelt sich um eine Anfangsposition eines Buckets in C_{T_1}.
$\Rightarrow lptr[merker] = i = lptr[2] = 2$

$i = 1$

Es handelt sich um eine Endposition eines Buckets in C_{T_1}.

$\Rightarrow merker = i = 1$

Es handelt sich um eine Anfangsposition eines Buckets in C_{T_1}.

$\Rightarrow lptr[merker] = i = lptr[1] = 1$

Für den Text T_1 ergibt sich folgendes Array *lptr*:

	1	2	3	4	5
$lptr_{T_1}$	1	2	3	-1	4

Der Ablauf des Algorithmus, um Array *lptr* für Text T_2 zu erstellen ist der folgende:

$i = 1$

Es handelt sich um eine Endposition eines Buckets in C_{T_2}.

$\Rightarrow merker = i = 1$

Es handelt sich um eine Anfangsposition eines Buckets in C_{T_2}.

$\Rightarrow lptr[merker] = i = lptr[1] = 1$

$i = 2$

Es handelt sich um eine Endposition eines Buckets in C_{T_2}.

$\Rightarrow merker = i = 2$

Es handelt sich um eine Anfangsposition eines Buckets in C_{T_2}.

$\Rightarrow lptr[merker] = i = lptr[2] = 2$

$i = 3$

Es handelt sich um eine Endposition eines Buckets in C_{T_2}.

$\Rightarrow merker = i = 3$

Es handelt sich um eine Anfangsposition eines Buckets in C_{T_2}.

$\Rightarrow lptr[merker] = i = lptr[3] = 3$

$i = 4$

Es handelt sich um eine Endposition eines Buckets in C_{T_2}.

$\Rightarrow merker = i = 4$

Es handelt sich um eine Anfangsposition eines Buckets in C_{T_2}.

$\Rightarrow lptr[merker] = i = lptr[4] = 4$

$i = 5$

Es handelt sich um eine Endposition eines Buckets in C_{T_2}.

$\Rightarrow merker = i = 5$

Es handelt sich um eine Anfangsposition eines Buckets in C_{T_2}.

$\Rightarrow lptr[merker] = i = lptr[5] = 5$

Für den Text T_2 ergibt sich folgendes Array *lptr*:

	1	2	3	4	5
$lptr_{T_2}$	1	2	3	4	5

Mit diesen beiden Hilfsarrays ist es nun möglich, die Sortierung durchzuführen, ohne auf Array A oder C zugreifen zu müssen.[1] Dazu werden die m Listen jeweils für Typ-S von links nach rechts oder für Typ-L von rechts nach links bucketweise durchlaufen. Für jedes Bucket der Listen werden zwei Schritte durchgeführt[2]:
(1) „Verschieben“ des betrachteten Suffixes an den aktuellen Anfang für Typ-S oder an das aktuelle Ende für Typ-L und Erhöhen des aktuellen Anfangs für Typ-S oder Erniedrigen des aktuellen Endes für Typ-L und (2) erneutes Durchlaufen des Buckets der Liste j und Überführen der betrachteten Suffixe in ein neues Bucket. Das bedeutet, während des Durchlaufs des Buckets erfolgen das „Verschieben“ des Suffixes und das Erhöhen oder Erniedrigen des aktuellen Anfangs oder des aktuellen Endes. Ist das Bucket vollständig durchlaufen, werden alle enthaltenen Suffixe in einem erneuten Durchlauf in ein neues Bucket überführt.
Für die Verarbeitung der Typ-S-Suffixe erfolgt der Zugriff auf das betrachtete Bucket über $R[i-j]$, wobei i der Inhalt der betrachteten Listenposition der Liste j ist.[3] Den aktuellen Anfang des Buckets erhält man über $lptr[R[i-j]]$. Das Erhöhen des aktuellen Anfangs erfolgt über $lptr[R[i-j]] = lptr[R[i-j]] + 1$. Das Überführen der Suffixe in das neue Bucket erfolgt durch $R[i-j] = lptr[R[i-j] - 1$ und

$$lptr[R[i-j]] = \begin{cases} R[i-j], & \text{wenn } lptr[R[i-j]] == -1, \\ lptr[R[i-j]] - 1, & \text{sonst.} \end{cases}$$

Für die Verarbeitung der Typ-L-Suffixe erfolgt der Zugriff auf das betrachtete Bucket über $R[i-j]$, wobei i der Inhalt der betrachteten Listenposition der Liste j ist. Das aktuelle Ende des Buckets erhält man über $lptr[R[i-j]]$. Das Erniedrigen des aktuellen Endes erfolgt über $lptr[R[i-j]] = lptr[R[i-j]] - 1$. Das Überführen der Suffixe in das neue Bucket erfolgt durch $R[i-j] = lptr[R[i-j] + 1$ und

$$lptr[R[i-j]] = \begin{cases} R[i-j], & \text{wenn } lptr[R[i-j]] == -1, \\ lptr[R[i-j]] + 1, & \text{sonst.} \end{cases}$$

1 Vgl. Ko u.a. (2005), S. 152.
2 Vgl. Ko u.a. (2005), S. 152.
3 Vgl. Ko u.a. (2005), S. 152.

Zusammengefasst als Algorithmus bedeutet das:

Algorithmus 16 sortiereSuffixeMitRUndLptr

Eingabe: Array R, Array $lptr$, $sortS$

Ausgabe: Array C, lexikografisch sortiert nach den ersten m Zeichen der Suffixe

```
1:  for j = 1, ..., m do
2:      if sortS == true then
3:          k = 1
4:      else
5:          k = Länge von Liste j
6:      end if
7:      merker = k
8:      while Liste j noch nicht vollständig durchlaufen do
9:          while Bucketgrenze nicht erreicht do
10:             Lies den Wert an Position k in Liste j aus, das ist i
11:             t = i − j
12:             Lies R[t] aus, das ist x
13:             Lies lptr[x] aus, das ist y
14:             if sortS == true then
15:                 lptr[x] = y + 1
16:                 k = k + 1
17:             else
18:                 lptr[x] = y − 1
19:                 k = k − 1
20:             end if
21:         end while
22:         k = merker
23:         while Bucketgrenze nicht erreicht do
24:             Lies den Wert an Position k in Liste j aus, das ist i
25:             t = i − j
26:             Lies R[t] aus, das ist x
27:             Lies lptr[x] aus, das ist y
28:             if sortS == true then
29:                 z = y − 1
30:                 R[t] = z
31:                 Lies lptr[z] aus, das ist q
```

Algorithmus 16 sortiereSuffixeMitRUndLptr (Teil 2)

```
32:                 if q == −1 then
33:                     lptr[z] = z
34:                 else
35:                     lptr[z] = q − 1
36:                 end if
37:                 k = k + 1
38:             else
39:                 z = y + 1
40:                 R[t] = z
41:                 Lies lptr[z] aus, das ist q
42:                 if q == −1 then
43:                     lptr[z] = z
44:                 else
45:                     lptr[z] = q + 1
46:                 end if
47:                 k = k − 1
48:             end if
49:         end while
50:         merker = k
51:     end while
52: end for
```

Für den Text T_1 sieht der Ablauf wie folgt aus:

j = 1; **k = 1**; **merker = 1**

Liste j noch nicht vollständig durchlaufen

Bucketgrenze noch nicht erreicht

$i = j[k] = j[1] = 3$

$t = i - j = 3 - 1 = 2$

Bucket auslesen: $x = R_{T_1}[t] = R_{T_1}[2] = 3$

aktuellen Bucketanfang auslesen: $y = lptr_{T_1}[x] = lptr_{T_1}[3] = 3$

$\Rightarrow$ aktuellen Anfang um Eins erhöhen: $lptr_{T_1}[3] = y + 1 = 4$

$k = k + 1 = 2$

Bucketgrenze noch nicht erreicht

$i = j[k] = j[2] = 8$

$t = i - j = 8 - 1 = 7$

Bucket auslesen: $x = R_{T_1}[t] = R_{T_1}[7] = 2$

aktuellen Bucketanfang auslesen: $y = lptr_{T_1}[x] = lptr_{T_1}[2] = 2$

$\Rightarrow$ aktuellen Anfang um Eins erhöhen: $lptr_{T_1}[2] = y + 1 = 3$

$k = k + 1 = 3$

Bucketgrenze erreicht

k = merker = 1

Bucketgrenze noch nicht erreicht

$i = j[k] = j[1] = 3$

$t = i - j = 3 - 1 = 2$

Bucket auslesen: $x = R_{T_1}[t] = R_{T_1}[2] = 3$

aktuellen Bucketanfang auslesen: $y = lptr_{T_1}[x] = lptr_{T_1}[3] = 4$

$z = y - 1 = 4 - 1 = 3$

$\Rightarrow$ neues Bucket zuweisen: $R_{T_1}[2] = z = 3$

$q = lptr_{T_1}[z] = lptr_{T_1}[3] = 4$

$\Rightarrow$ Bucketanfang vorhanden,
deshalb Bucketanfang um Eins verringern:

$lptr_{T_1}[3] = q - 1 = 3$

$k = k + 1 = 2$

Bucketgrenze noch nicht erreicht

$i = j[k] = j[2] = 8$

$t = i - j = 8 - 1 = 7$

Bucket auslesen: $x = R_{T_1}[t] = R_{T_1}[7] = 2$

aktuellen Bucketanfang auslesen: $y = lptr_{T_1}[x] = lptr_{T_1}[2] = 3$

$z = y - 1 = 3 - 1 = 2$

$\Rightarrow$ neues Bucket zuweisen: $R_{T_1}[7] = z = 2$

$q = lptr_{T_1}[z] = lptr_{T_1}[2] = 3$

$\Rightarrow$ Bucketanfang vorhanden,
deshalb Bucketanfang um Eins verringern:

$lptr_{T_1}[3] = q - 1 = 2$

$k = k + 1 = 3$

Bucketgrenze erreicht

merker = k = 3

Liste j noch nicht vollständig durchlaufen

Bucketgrenze noch nicht erreicht

$i = j[k] = j[3] = 4$

$t = i - j = 4 - 1 = 3$

Bucket auslesen: $x = R_{T_1}[t] = R_{T_1}[3] = 5$

aktuellen Bucketanfang auslesen: $y = lptr_{T_1}[x] = lptr_{T_1}[5] = 4$

$\Rightarrow$ aktuellen Anfang um Eins erhöhen: $lptr_{T_1}[5] = y + 1 = 5$

$k = k + 1 = 4$

Bucketgrenze noch nicht erreicht

$i = j[k] = j[4] = 9$

$t = i - j = 9 - 1 = 8$

Bucket auslesen: $x = R_{T_1}[t] = R_{T_1}[8] = 5$

aktuellen Bucketanfang auslesen: $y = lptr_{T_1}[x] = lptr_{T_1}[5] = 5$

$\Rightarrow$ aktuellen Anfang um Eins erhöhen: $lptr_{T_1}[5] = y + 1 = 6$

$k = k + 1 = 5$

Bucketgrenze erreicht

k = merker = 3

Bucketgrenze noch nicht erreicht

$i = j[k] = j[3] = 4$

$t = i - j = 4 - 1 = 3$

Bucket auslesen: $x = R_{T_1}[t] = R_{T_1}[3] = 5$

aktuellen Bucketanfang auslesen: $y = lptr_{T_1}[x] = lptr_{T_1}[5] = 6$

$z = y - 1 = 6 - 1 = 5$

$\Rightarrow$ neues Bucket zuweisen: $R_{T_1}[3] = z = 5$

$q = lptr_{T_1}[z] = lptr_{T_1}[5] = 6$

$\Rightarrow$ Bucketanfang vorhanden,
deshalb Bucketanfang um Eins verringern:

$lptr_{T_1}[4] = q - 1 = 5$

$k = k + 1 = 4$

Bucketgrenze noch nicht erreicht

$i = j[k] = j[4] = 9$

$t = i - j = 9 - 1 = 8$

Bucket auslesen: $x = R_{T_1}[t] = R_{T_1}[8] = 5$

aktuellen Bucketanfang auslesen: $y = lptr_{T_1}[x] = lptr_{T_1}[5] = 5$

$z = y - 1 = 5 - 1 = 4$

$\Rightarrow$ neues Bucket zuweisen: $R_{T_1}[8] = z = 4$

$q = lptr_{T_1}[z] = lptr_{T_1}[4] = -1$

$\Rightarrow$ kein Bucketanfang vorhanden,
deshalb Bucketende als Bucketanfang zuweisen:

$lptr_{T_1}[4] = z = 4$

$k = k + 1 = 5$

Bucketgrenze erreicht
merker = k = 5

Liste j vollständig durchlaufen

Beim Durchlaufen der Listen 2 bis 4 ergeben sich keine Änderungen mehr. Die beiden Zusatzarrays sehen danach wie folgt aus:

	1	2	3	4	5	6	7	8	9	10
R_{T_1}	-1	3	5	-1	-1	-1	2	4	-1	1

	1	2	3	4	5
$lptr_{T_1}$	1	2	3	4	5

Aus diesen beiden Arrays kann die jeweilige Position der Suffixe in Array C rekonstruiert werden. Dafür liest man für jedes Suffix im Fall der Typ-S-Suffixe die Anfangsposition seines Buckets aus Array *lptr* aus und die Endposition seines Buckets aus R aus. Für die Typ-L-Suffixe ist die Anfangsposition in Array R vorhanden und die Endposition in Array *lptr*. Existiert nur noch ein Suffix pro Bucket, so sind die Suffixe in Array C lexikografisch sortiert und die Bucketangaben in Array *lptr* und in R sind eindeutig.
Betrachtet man Array C_{T_1} wie es bis zu diesem Verarbeitungsschritt noch „aussieht“, siehe S. 117, so erkennt man, dass sich lediglich die Suffixe an den Indizes 4 und 5 in einem gemeinsamen Bucket befinden. Also müssen auch nur diese evtl. in ihrer Position in Array C_{T_1} verändert werden. Exemplarisch wird im Folgenden die Positionsveränderung für Suffix $T_1[8]$ gezeigt:
Suche die Endposition von Suffix $T_1[8]$ indem $R_{T_1}[8]$ ausgelesen wird, das ist 4.
Suche die Anfangsposition von Suffix $T_1[8]$ indem $lptr_{T_1}[R_{T_1}[8]] = lptr_{T_1}[4]$ ausgelesen wird, das ist 4.

⇒ Die neue Position von Suffix $T_1[8]$ ist Index 4 in Array C_{T_1}, also $C_{T_1} = 8$.
Array C_{T_1} sieht also nach beiden erfolgten Positionsveränderungen wie folgt aus[1]:

	1	2	3	4	5
C_{T_1}	10	7	2	8	3

1 Wie in Ko u.a. (2005), S. 152 erläutert, ist es unnötig, das eigentliche Verschieben oder Vertauschen der Suffixe in Array C durchzuführen. Um die Erläuterungen aber besser nachvollziehbar zu gestalten, wird hier das Array C mit den explizit durchgeführten Vertauschungen angegeben.

Das bedeutet, nach der Verarbeitung der ersten der m Listen sind die Typ-S-Suffixe bereits sortiert. Feststellen lässt sich das am *lptr*-Array: Sind alle Einträge unterschiedlich und ungleich -1, so sind die Suffixe sortiert.
Für das Typ-L-Beispiel gilt diese Voraussetzung bereits bevor die Listen zur Sortierung herangezogen wurden. Dieser Schritt entfällt hier also.[1]

3.1.4.3.2.6 Konstruktion einer neuen Zeichenkette T' aufgrund der zu sortierenden Suffixe

Tritt der Fall ein, dass eine Typ-S- oder eine Typ-L-Teilzeichenkette Präfix eines anderen Typ-S- oder L-Suffixes ist, so kann es vorkommen, dass das Array C an diesem Punkt noch nicht lexikografisch sortiert ist.[2] Dann muss eine neue Zeichenkette T' erzeugt werden, die zu allen Typ-S- oder Typ-L-Teilzeichenketten korrespondiert.[3] Dafür werden die Teilzeichenketten durch ihre zugehörige Bucketnummer aus C ersetzt und T' ist dann die Sequenz der Bucketnummern in der Reihenfolge, in der sie in T auftauchen. Wird nun T' sortiert, siehe Kapitel 3.1.4.3.2.7, S. 132 ff., so sind anschließend auch die entsprechenden Suffixe aus T sortiert.[4]
Als Algorithmus kann das wie folgt dargestellt werden:

Algorithmus 17 Konstruktion einer Zeichenkette T'

Eingabe: Array C mit Buckets, nachdem alle Listen verarbeitet wurden

Ausgabe: Array C mit den sortierten Typ-S- oder Typ-L-Suffixen

1: Nummeriere alle vorhandenen Buckets in Array C aufsteigend bei 1 beginnend
2: Sortiere die in C enthaltenen Textpositionen aufsteigend, das ist die Zeichenkette x
3: Erzeuge T' als leere Zeichenkette
4: **for all** Textpositionen in x **do**
5: Ergänze T' um die zugehörige Bucketnummer der aktuellen Textposition
6: **end for**

Problematisch wird die Nummerierung, wie sie in Algorithmus 17 Zeile 1 durchgeführt wird, bei längeren Texten und entsprechend längeren Präfixen, da dann mehrstellige Bucketnummern auftreten. Das muss dann entsprechend der verwendeten Programmiersprache gelöst werden. In dieser Arbeit wird die Konstruktion

1 Im Anhang der vorliegenden Arbeit ist beispielhaft die Verarbeitung der ersten der m Listen für das Typ-L-Beispiel aufgeführt, um die Unterschiede in der Verarbeitung zu den Typ-S-Suffixen zu verdeutlichen, siehe S. 583 ff.
2 Vgl. Ko u.a. (2003), S. 206; Ko u.a. (2005), S. 149.
3 Vgl. Ko u.a. (2003), S. 205 f.; Ko u.a. (2005), S. 148 f.
4 Vgl. Ko u.a. (2003), S. 205-207; Ko u.a. (2005), S. 148-150.

von T' in der Implementierung in Kapitel 3.1.4.4.2.2, ab S. 174, erläutert. Da beide Beispiele bereits sortiert sind, entfällt dieser Verarbeitungsschritt für die vorher betrachteten Beispiele. Um dennoch anhand eines Beispiels diese Verarbeitung zeigen zu können, wird die Zeichenkette T =„abrakadabra$“ verwendet. Man würde hier die Typ-L-Suffixe sortieren, da sie in geringerer Anzahl vorhanden sind als die Typ-S-Suffixe. Dann würde aber der Fall der Erzeugung von T' nicht auftreten. Aus diesem Grund wird die Verarbeitung gezeigt, wenn die Typ-S-Suffixe sortiert werden sollen.
Als kurzer Überblick sind im Folgenden die benötigten Arrays Typ_T, A_T, $Dist_T$, C_T, R_T und $lptr_T$ angegeben sowie die benötigten Listen.

	1	2	3	4	5	6	7	8	9	10	11	12
T	a	b	r	a	k	a	d	a	b	r	a	$
Typ_T	S	S	L	S	L	S	L	S	S	L	L	S/L

	1	2	3	4	5	6	7	8	9	10	11	12
A_T	12	1	4	6	8	11	2	9	7	5	3	10
S-Dist	0	1	1	2	1	2	1	2	1	1	2	3

	1	2	3	4	5	6	7
C_T	12	1	4	6	8	2	9

	1	2	3	4	5	6	7	8	9	10	11	12
R_T	5	7	-1	5	-1	5	-1	5	7	-1	-1	1

	1	2	3	4	5	6	7
$lptr_T$	1	-1	-1	-1	2	-1	6

Die Listen mit Buckets sehen wie folgt aus:

1:	2	9	7	5	3	10
2:	4	6	8	11		
3:	12					

Die Arrays C_T, R_T und $lptr_T$ sehen nach dem Durchlauf der ersten Liste nach Algorithmus 16 wie folgt aus:

	1	2	3	4	5	6	7
C_T	12	1	8	6	4	9	2

	1	2	3	4	5	6	7	8	9	10	11	12
R_T	3	7	-1	5	-1	4	-1	3	6	-1	-1	1

	1	2	3	4	5	6	7
$lptr_T$	1	-1	2	4	5	6	7

Die neuen Buckets in Array C_T und die Änderung der Sortierung der Textpositionen ergeben sich aus den Änderungen in Array R_T und Array $lptr_T$. Das Verfahren des Ablesens ist auf S. 128 beschrieben. Diese Änderungen am Array C_T werden in der späteren Implementierung nicht explizit vorgenommen, sondern hier nur der besseren Übersichtlichkeit wegen aufgeführt. Die Durchläufe durch die zweite und dritte Liste führen nicht zu Änderungen der Arrays. Erkennbar ist in Array C_T, dass sich die Textpositionen 1 und 8 im gleichen Bucket befinden, d.h., noch nicht lexikografisch eindeutig sortiert sind. Verdeutlichen lässt sich das durch die Angabe der dazugehörigen Zeichenkettenausschnitte unter den Textpositionen. Durch die Listen erfolgte eine Sortierung nach den ersten 4 Zeichen des jeweiligen Suffixes, da $m = 3$ Listen vorhanden waren. Die nachfolgenden Zeichen müssen sich jedoch an irgendeiner Stelle voneinander unterscheiden. Also führt die bisherige Sortierung von Array C_T noch nicht zu einer abschließenden Sortierung der Typ-S-Suffixe.

	1	2	3	4	5	6	7
C_T	12	1	8	6	4	9	2
	$	a	a	a	a	b	b
		b	b	d	k	r	r
		r	r	a	a	a	a
		a	a	b	d	$	k

Um Array C_T vollständig sortieren zu können, wird eine neue Zeichenkette T' aus den Bucketnummern des Arrays C_T, wie in Algorithmus 17 vorgegeben, erstellt.
Eine Nummerierung der Buckets im Beispiel sieht wie folgt aus:

	1	2	3	4	5	6	7
C_T	12	1	8	6	4	9	2
	$	a	a	a	a	b	b
		b	b	d	k	r	r
		r	r	a	a	a	a
		a	a	b	d	$	k
Bucketnr.	1	2		3	4	5	6

Sortiert man dem Algorithmus entsprechend nun die Textpositionen aufsteigend und hängt jeweils die nächste Bucketnummer an die Zeichenkette T', so ergibt sich der folgende Ablauf:

Textposition 1 ist in Bucket 2 $\Rightarrow T' =$“2“
Textposition 2 ist in Bucket 6 $\Rightarrow T' =$“26“
Textposition 4 ist in Bucket 4 $\Rightarrow T' =$“264“
Textposition 6 ist in Bucket 3 $\Rightarrow T' =$“2643“
Textposition 8 ist in Bucket 2 $\Rightarrow T' =$“26432“
Textposition 9 ist in Bucket 5 $\Rightarrow T' =$“264325“
Textposition 12 ist in Bucket 1 $\Rightarrow T' =$“2643251“

Die neue zu sortierende Zeichenkette für das Beispiel ist also: $T' = 2643251$.

3.1.4.3.2.7 Rekursive Anwendung des Gesamtalgorithmus auf die Zeichenkette T’

Der zuvor beschriebene Gesamtalgorithmus 6 wird rekursiv auf T' angewendet, um letztendlich die Sortierung der Typ-S- oder Typ-L-Suffixe zu erhalten.[1]
Für das Beispiel $T =$„abrakadabra$“ wurde die Zeichenkette $T' = 2643251$ erzeugt. An diese wird als erstes ebenfalls das „$“ als eindeutiges Endezeichen angehängt. Zusätzlich werden die Typen der Suffixe laut Algorithmus 7 bestimmt. Es ergibt sich also Folgendes:

	1	2	3	4	5	6	7	8
T'	2	6	4	3	2	5	1	$
$\text{Typ}_{T'}$	S	L	L	L	S	L	L	S/L

Das Array $A'_{T'}$ wird durch Algorithmus 8 erzeugt. Die Erzeugung der m' Listen und die Berechnung der entsprechenden S-Distanzen erfolgt wie in Algorithmus 9 beschrieben.

	1	2	3	4	5	6	7	8
$A'_{T'}$	8	7	1	5	4	3	6	2
$Dist_{T'}$	0	1	2	3	4	1	2	3

1 Vgl. Ko u.a. (2003), S. 205; Ko u.a. (2005), S. 148.

1:	6	2
2:	7	3
3:	8	4
4:	5	

Das Array $C'_{T'}$, das sich laut Algorithmus 12 ergibt, ist das folgende:

	1	2	3
$C'_{T'}$	8	1	5

Die Arrays $R'_{T'}$ und $lptr'_{T'}$ werden durch die Algorithmen 14 und 15 erstellt.

	1	2	3	4	5	6	7	8
$R'_{T'}$	3	-1	-1	-1	3	-1	-1	1

	1	2	3
$lptr'_{T'}$	1	-1	2

Führt man einen Durchlauf durch die erste Liste durch, ergeben sich folgende Änderungen in den Arrays $R'_{T'}$ und $lptr'_{T'}$:

	1	2	3	4	5	6	7	8
$R'_{T'}$	3	-1	-1	-1	2	-1	-1	1

	1	2	3
$lptr'_{T'}$	1	2	3

Das bedeutet, bereits nach dem Durchlauf der ersten Liste sind die Suffixe im Array $C'_{T'}$ sortiert, da jede Textposition sich in einem eigenen Bucket befindet. Die Arrays $R'_{T'}$ und $lptr'_{T'}$ geben an, dass die Sortierung der Textpositionen wie folgt ist:

	1	2	3
$C'_{T'}$	8	5	1

Wird das Array $C'_{T'}$ jetzt dazu benutzt, das Array $A'_{T'}$ zu sortieren[1], so ergibt sich das folgende Array $A'_{T'}$.

	1	2	3	4	5	6	7	8
$A'_{T'}$	8	7	5	1	4	3	6	2

1 Eine genaue Beschreibung dieses Ablaufs befindet sich in Kapitel 3.1.4.3.2.8, S. 135 ff.

Aus dem Array $A'_{T'}$ kann abgelesen werden, dass sich die Textposition 5 lexikografisch vor der Textposition 1 befindet. Diese Textpositionen beinhalten beide das Zeichen 2 in der Zeichenkette T', korrespondieren aber im Array C_T, welches sich auf T bezieht, mit unterschiedlichen Textpositionen. Um festzustellen, mit welchen der Textpositionen aus C_T die Textpositionen aus $C'_{T'}$ korrespondieren, muss man sich die entsprechenden Suffixe als Ganzes anschauen:
Textposition 5 im Array $A'_{T'}$ steht vor Textposition 1 im Array $A'_{T'}$:
$\Rightarrow$ Suffix 251 aus T' ist lexikografisch kleiner als Suffix 2643251 aus T'.
$\Rightarrow$ Ersetze die Reihenfolge der Bucketnummern, die der Inhalt der Suffixe von T' sind, durch die einzigen möglichen Textpositionen: 2 korrespondiert zu 1 oder 8. Nach der 2 steht die 5, die mit 9 korrespondiert. Da, um T' zu bilden, eine aufsteigende Sortierung der Textpositionen aus C vorgenommen wurde, kann sich die 2 in diesem Fall nur auf die 8 beziehen, da sich die nachfolgende 5 auf die 9 bezieht und die 8 näher zur 9 liegt als die 1, oder anders ausgedrückt: zwischen der 1 und der nachfolgenden 9 liegen noch andere im Array C_T enthaltene Textpositionen, so dass die Reihenfolge $2;5$ aus T' sich nur auf die Textpositionen $8;9$ im Array C_T beziehen kann.
Die gleiche Argumentation wird für die Korrespondenz zwischen dem Suffix 2643251 aus T' und der 1 aus dem Array C_T angewendet: die Reihenfolge $2;6$ kann sich nur auf die Textpositionen $1;2$ aus dem Array C_T beziehen, da 2 aus T' mit 1 oder 8 aus dem Array C_T korrespondiert und die 6 mit 2 aus dem Array C_T. Würde man annehmen, dass sich die 2 aus T' auf die 8 aus dem Array C_T bezieht, so müsste die Textposition 8 bei aufsteigender Sortierung der Textpositionen aus dem Array C_T vor der Textposition 2 liegen. Das ist nicht möglich, daher korrespondiert das Suffix 2643251 aus T' mit dem Suffix aus T, das mit der Textposition 1 in T beginnt.
Folglich liegt im zu sortierenden Array C_T die Textposition 8 vor der Textposition 1. Da das die einzige Stelle ist, die zuvor noch nicht sortiert war, ist das die einzige Änderung an Array C_T, die durchgeführt werden muss, um es zur abschließenden Sortierung von Array A_T nutzen zu können.
Der Algorithmus, um die korrespondierenden Suffixe zu finden, sieht wie folgt aus:

Algorithmus 18 rekonstruiereCAusSortiertemT'

Eingabe: Array C, Text T, Array A', Text T'

Ausgabe: lexikografisch sortiertes Array C

1: $pos = 1$
2: **for** $i = 2, ..., A'.length$ **do**
3: Lies $A'[i]$ aus, das ist x
4: Lies $T'[x]$ aus, das ist t

Algorithmus 18 rekonstruiereCAusSortiertemT' (Teil 2)

```
5:    Entferne das Endezeichen von t
6:    Lies die Textpositionen aus Bucket t[1] aus C aus, das ist y
7:    if Länge von y > 1 then
8:        Entscheide anhand des Abstands zur nächsten eindeutigen
          Bucketnummer aus t, welche der Textpositionen aus y mit t[1]
          korrespondiert, diese Textposition ist textPos
9:    else
10:       textPos = y
11:   end if
12:   C[pos] = textPos
13:   pos = pos + 1
14: end for
```

3.1.4.3.2.8 Konstruktion des Suffix Arrays aufgrund der Vorsortierung der Typ-L- oder Typ-S-Suffixe

Die benötigte Bucketierung des Arrays A nach den ersten Zeichen der Suffixe ist bereits erfolgt.[1] Die Sortierung des kleineren Teils der Suffixe ist mit den vorangegangenen Schritten ebenfalls abgeschlossen, so dass Array C die Sortierung dieser Suffixe widerspiegelt. Mit Hilfe des Arrays C lässt sich das Array A abschließend sortieren, so dass es das Suffix Array des entsprechenden Textes ist.[2] Als Algorithmus zur abschließenden Sortierung ergibt sich der folgende:

Algorithmus 19 Algorithmus zur abschließenden Sortierung des Arrays A

Eingabe: Array A mit Buckets aufgrund des ersten Zeichens, Array C, $sortS$, Text T, Array Typ

Ausgabe: Suffix Array A für den Text T

```
1: Erzeuge Array Rev
2: for i = 1, ..., A.length do
3:     Lies A[i] aus, das ist x
4:     Rev[x] = i
5: end for
6: if sortS == true then
7:     for i = C.length, ..., 1 do
8:         Lies C[i] aus, das ist x
```

1 Siehe S. 109 der vorliegenden Arbeit.
2 Vgl. Ko u.a. (2003), S. 203 f.; Ko u.a. (2005), S. 146-148.

Algorithmus 19 Algorithmus zur abschließenden Sortierung des Arrays A (Teil 2)

```
9:          Lies die Position des Suffixes x in A aus, das ist y = Rev[x]
10:         Lies die Endposition des Buckets von A[y] aus, das ist z
11:         if y ≠ z then
12:             w = A[y]
13:             A[y] = A[z]
14:             Rev[z] = y
15:             A[z] = w
16:             Rev[w] = z
17:         end if
18:         Verringere die Endposition des Buckets von A[y] um 1
19:     end for
20:     for i = 1, ..., A.length do
21:         Lies A[i] aus, das ist x
22:         y = x − 1
23:         if y > 0 then
24:             Lies Typ[y] aus, das ist z
25:             if z == L then
26:                 Lies die Position des Suffixes y in A aus, das ist pos = Rev[y]
27:                 Lies die Anfangsposition des Buckets von A[pos] aus, das ist q
28:                 if q ≠ pos then
29:                     w = A[q]
30:                     v = A[pos]
31:                     A[q] = A[pos]
32:                     Rev[v] = q
33:                     A[pos] = w
34:                     Rev[w] = pos
35:                 end if
36:                 Erhöhe die Anfangsposition des Buckets von A[pos] um 1
37:             end if
38:         end if
39:     end for
40: else
41:     for i = 1, ..., C.length do
42:         Lies C[i] aus, das ist x
43:         Lies die Position des Suffixes x in A aus, das ist y = Rev[x]
44:         Lies die Anfangsposition des Buckets von A[y] aus, das ist z
45:         if y ≠ z then
```

Algorithmus 19 Algorithmus zur abschließenden Sortierung des Arrays A (Teil 3)

```
46:             w = A[z]
47:             A[z] = A[y]
48:             Rev[x] = z
49:             A[y] = w
50:             Rev[w] = y
51:         end if
52:         Erhöhe die Anfangsposition des Buckets von A[y] um 1
53:     end for
54:     for i = A.length, ..., 1 do
55:         Lies A[i] aus, das ist x
56:         y = x − 1
57:         if y > 0 then
58:             Lies Typ[y] aus, das ist z
59:             if z == S then
60:                 Lies die Position des Suffixes y in A aus, das ist pos = Rev[y]
61:                 Lies die Endposition des Buckets von A[pos] aus, das ist q
62:                 if q ≠ pos then
63:                     w = A[q]
64:                     v = A[pos]
65:                     A[q] = A[pos]
66:                     Rev[v] = q
67:                     A[pos] = w
68:                     Rev[w] = pos
69:                 end if
70:                 Verringere die Endposition des Buckets von A[pos] um 1
71:             end if
72:         end if
73:     end for
74: end if
```

Zunächst wird für jedes Array A der beiden Beispieltexte T_1 und T_2 ein Array Rev erzeugt.[1] Diese Zusatzarrays speichern die Positionen der Suffixe in Array A, um einen direkten Zugriff auf die Suffixe in Array A zu ermöglichen. Wie man später sehen wird, betrachtet man die Textpositionen, an denen die entsprechenden Suffixe beginnen. Um einen direkten Zugriff in Array A auf diese Textpositionen zu

1 Vgl. Ko u.a. (2003), S. 203.

ermöglichen, fehlt jedoch die Information, wo sie sich aufgrund ihrer lexikografischen Sortierung in Array A befinden. Diese Information erhält man durch die Arrays Rev. Sie sehen für die beiden Beispieltexte wie folgt aus:

	1	2	3	4	5	6	7	8	9	10
Rev_{T_1}	6	3	7	9	5	4	2	8	10	1
Rev_{T_2}	7	8	3	4	6	5	2	9	10	1

Es ist nun möglich, die Position eines Suffix in Array A direkt zu bestimmen. Zum Beispiel befindet sich das Suffix, das an Textposition 4 im Text T_1 beginnt, in Array A_{T_1} an Position 9. Das Suffix, das an Textposition 4 im Text T_2 beginnt, befindet sich in Array A_{T_2} an Position 4.

Im Fall der Typ-S-Suffixe wird Array C von rechts nach links durchlaufen und die entsprechenden Suffixe in Array A werden an das aktuelle Ende ihres dortigen Buckets getauscht.[1] Das aktuelle Ende wird dann um eins nach links verschoben. Für das Beispiel ergibt sich folgende Verarbeitung[2]:

	1	2	3	4	5
C_{T_1}	10	7	2	8	3

	1	2	3	4	5	6	7	8	9	10
A_{T_1}	10	7	2	6	5	1	3	8	4	9

$i = 5$
$x = C_{T_1}[i] = C_{T_1}[5] = 3$
$y = Rev_{T_1}[x] = Rev_{T_1}[3] = 7$
$z =$ Endposition des Buckets von $A_{T_1}[y] = A_{T_1}[7] = 8$
$y \neq z$: Suffix 3 an die Endposition tauschen
$w = A_{T_1}[y] = A_{T_1}[7] = 3$
$A_{T_1}[7] = A_{T_1}[z] = A_{T_1}[8] = 8$
$Rev_{T_1}[8] = 7$
$A_{T_1}[8] = w = 3$
$Rev_{T_1}[3] = 8$
Endposition des gerade betrachteten Buckets um 1 verringern,
d.h., Endposition ist 7[3]

1 Vgl. Ko u.a. (2003), S. 203; Ko u.a. (2005), S. 147.
2 Zur Veranschaulichung sind die bereits erzeugten Arrays C_{T_1} und A_{T_1} nochmals abgebildet.
3 In der nachfolgenden Tabelle sind die Textpositionen fett und unterstrichen dargestellt, die vertauscht wurden.

Die Arrays A_{T_1} und Rev_{T_1} sehen nach dem Vertauschen wie folgt aus:

	1	2	3	4	5	6	7	8	9	10
A_{T_1}	10	7	2	6	5	1	**8**	**3**	4	9
Rev_{T_1}	6	3	**8**	9	5	4	2	**7**	10	1

$i = 4$
$x = C_{T_1}[i] = C_{T_1}[4] = 8$
$y = Rev_{T_1}[x] = Rev_{T_1}[8] = 7$
$z =$ Endposition des Buckets von $A_{T_1}[y] = A_{T_1}[7] = 7$
$y = z:$ Suffix 8 steht bereits an der Endposition
Endposition des gerade betrachteten Buckets um 1 verringern,
d.h., Endposition ist 6
$i = 3$
$x = C_{T_1}[i] = C_{T_1}[3] = 2$
$y = Rev_{T_1}[x] = Rev_{T_1}[2] = 3$
$z =$ Endposition des Buckets von $A_{T_1}[y] = A_{T_1}[3] = 3$
$y = z:$ Suffix 2 steht bereits an der Endposition
Endposition des gerade betrachteten Buckets um 1 verringern,
d.h., Endposition ist 2
$i = 2$
$x = C_{T_1}[i] = C_{T_1}[2] = 7$
$y = Rev_{T_1}[x] = Rev_{T_1}[7] = 2$
$z =$ Endposition des Buckets von $A_{T_1}[y] = A_{T_1}[2] = 2$
$y = z:$ Suffix 7 steht bereits an der Endposition
Endposition des gerade betrachteten Buckets um 1 verringern,
d.h., Endposition ist 1
$i = 1$
$x = C_{T_1}[i] = C_{T_1}[1] = 10$
$y = Rev_{T_1}[x] = Rev_{T_1}[10] = 1$
$z =$ Endposition des Buckets von $A_{T_1}[y] = A_{T_1}[1] = 1$
$y = z:$ Suffix 10 steht bereits an der Endposition
Endposition des gerade betrachteten Buckets um 1 verringern,
d.h., Endposition ist 0[1]

1 Diese Position existiert nicht im Array. In einer Implementierung müsste das erkannt und verhindert werden, dass auf eine solche Position zugegriffen wird.

Danach sieht Array A_{T_1} wie folgt aus:

	1	2	3	4	5	6	7	8	9	10
A_{T_1}	10	7	2	6	5	1	8	3	4	9

Alle Typ-S-Suffixe befinden sich jetzt auf ihrer korrekten Position in Array A_{T_1}.
Im Fall der Typ-L-Suffixe wird Array C_{T_2} von links nach rechts durchlaufen und die entsprechenden Suffixe werden in Array A_{T_2} an den aktuellen Anfang ihres dortigen Buckets getauscht.[1] Der aktuelle Anfang wird dann um eins nach rechts verschoben. Für das Beispiel ergibt sich folgende Verarbeitung[2]:

	1	2	3	4	5
C_{T_2}	10	6	5	2	9

	1	2	3	4	5	6	7	8	9	10
A_{T_2}	10	7	3	4	6	5	1	2	8	9

$i = 1$
$x = C_{T_2}[i] = C_{T_2}[1] = 10$
$y = Rev_{T_2}[x] = Rev_{T_2}[10] = 1$
$z =$ Anfangsposition des Buckets von $A_{T_2}[y] = A_{T_2}[1] = 1$
$y = z$: Suffix 10 steht bereits an der Anfangsposition
Anfangsposition des gerade betrachteten Buckets um 1 erhöhen,
d.h., Anfangsposition ist 2
$i = 2$
$x = C_{T_2}[i] = C_{T_2}[2] = 6$
$y = Rev_{T_2}[x] = Rev_{T_2}[6] = 5$
$z =$ Anfangsposition des Buckets von $A_{T_2}[y] = A_{T_2}[5] = 5$
$y = z$: Suffix 6 steht bereits an der Anfangsposition
Anfangsposition des gerade betrachteten Buckets um 1 erhöhen,
d.h., Anfangsposition ist 6
$i = 2$
$x = C_{T_2}[i] = C_{T_2}[2] = 6$
$y = Rev_{T_2}[x] = Rev_{T_2}[6] = 5$
$z =$ Anfangsposition des Buckets von $A_{T_2}[y] = A_{T_2}[5] = 5$
$y = z$: Suffix 6 steht bereits an der Anfangsposition

1 Vgl. Ko u.a. (2003), S. 205; Ko u.a. (2005), S. 148.
2 Zur Veranschaulichung sind die bereits erzeugten Arrays C_{T_2} und A_{T_2} nochmal abgebildet.

Anfangsposition des gerade betrachteten Buckets um 1 erhöhen,
d.h., Anfangsposition ist 6
$i = 3$
$x = C_{T_2}[i] = C_{T_2}[3] = 5$
$y = Rev_{T_2}[x] = Rev_{T_2}[5] = 6$
$z =$ Anfangsposition des Buckets von $A_{T_2}[y] = A_{T_2}[6] = 6$
$y = z$: Suffix 5 steht bereits an der Anfangsposition
Anfangsposition des gerade betrachteten Buckets um 1 erhöhen,
d.h., Anfangsposition ist 7
$i = 4$
$x = C_{T_2}[i] = C_{T_2}[4] = 2$
$y = Rev_{T_2}[x] = Rev_{T_2}[2] = 8$
$z =$ Anfangsposition des Buckets von $A_{T_2}[y] = A_{T_2}[8] = 8$
$y = z$: Suffix 2 steht bereits an der Anfangsposition
Anfangsposition des gerade betrachteten Buckets um 1 erhöhen,
d.h., Anfangsposition ist 9
$i = 5$
$x = C_{T_2}[i] = C_{T_2}[5] = 9$
$y = Rev_{T_2}[x] = Rev_{T_2}[9] = 10$
$z =$ Anfangsposition des Buckets von $A_{T_2}[y] = A_{T_2}[10] = 10$
$y = z$: Suffix 9 steht bereits an der Anfangsposition
Anfangsposition des gerade betrachteten Buckets um 1 erhöhen,
d.h., Anfangsposition ist 11[1]
Danach sieht Array A_{T_2} wie folgt aus:

	1	2	3	4	5	6	7	8	9	10
A_{T_2}	10	7	3	4	6	5	1	2	8	9

Alle Typ-L-Suffixe befinden sich jetzt auf ihrer korrekten Position.
Im Fall der Typ-S-Suffixe wird abschließend das Array A_{T_1} von links nach rechts durchlaufen.[2] Wird $A_{T_1}[i]$ betrachtet und handelt es sich bei $A_{T_1}[i] - 1$ um ein Typ-L-Suffix, so wird $A_{T_1}[i] - 1$ an den aktuellen Anfang seines Buckets in A_{T_1} getauscht und der Anfang um eins nach rechts verschoben.

1 Es gilt das Gleiche, wie beim Beispiel für Text T_1 gesagt: Es existiert keine Position 11 im Array A_{T_2}. In einer Implementierung müsste das erkannt und verhindert werden, dass auf eine solche Position zugegriffen wird.
2 Vgl. Ko u.a. (2003), S. 203; Ko u.a. (2005), S. 147.

Für das Beispiel sieht der Ablauf wie folgt aus:

	1	2	3	4	5	6	7	8	9	10
A_{T_1}	10	7	2	6	5	1	8	3	4	9

$i = 1$
$x = A_{T_1}[i] = A_{T_1}[1] = 10$
$y = x - 1 = 10 - 1 = 9$
$z = Typ\ T1[y] = Typ\ T1[9] = L$
$pos = Rev_{T_1}[y] = Rev_{T_1}[9] = 10$
$q =$ Anfangsposition des Buckets von $A_{T_1}[pos] = A_{T_1}[10] = 9$
$q \neq pos:$ Suffix 9 an die Anfangsposition tauschen
$w = A_{T_1}[q] = A_{T_1}[9] = 4$
$v = A_{T_1}[pos] = A_{T_1}[10] = 9$
$A_{T_1}[9] = A_{T_1}[pos] = A_{T_1}[10] = 9$
$Rev_{T_1}[9] = 9$
$A_{T_1}[10] = w = 4$
$Rev_{T_1}[4] = 10$
Anfangsposition des gerade betrachteten Buckets um 1 erhöhen,
d.h., Anfangsposition ist 10
Nach diesem Schritt sehen die beiden Arrays A und Rev wie folgt aus:

	1	2	3	4	5	6	7	8	9	10
A_{T_1}	10	7	2	6	5	1	8	3	**9**	**4**
Rev_{T_1}	6	3	8	**10**	5	4	2	7	**9**	1

$i = 2$
$x = A_{T_1}[i] = A_{T_1}[2] = 7$
$y = x - 1 = 7 - 1 = 6$
$z = Typ_{T_1}[y] = Typ_{T_1}[6] = L$
$pos = Rev_{T_1}[y] = Rev_{T_1}[6] = 4$
$q =$ Anfangsposition des Buckets von $A_{T_1}[pos] = A_{T_1}[4] = 4$
$q = pos:$ Suffix 6 ist an der Anfangsposition
Anfangsposition des gerade betrachteten Buckets um 1 erhöhen,
d.h., Anfangsposition ist 5
$i = 3$
$x = A_{T_1}[i] = A_{T_1}[3] = 2$
$y = x - 1 = 2 - 1 = 1$
$z = Typ_{T_1}[y] = Typ_{T_1}[1] = L$

$pos = Rev_{T_1}[y] = Rev_{T_1}[1] = 6$
$q =$ Anfangsposition des Buckets von $A_{T_1}[pos] = A_{T_1}[6] = 6$
$q = pos$: Suffix 1 ist an der Anfangsposition
Anfangsposition des gerade betrachteten Buckets um 1 erhöhen,
d.h., Anfangsposition ist 7
$i = 4$
$x = A_{T_1}[i] = A_{T_1}[4] = 6$
$y = x - 1 = 6 - 1 = 5$
$z = Typ_{T_1}[y] = Typ_{T_1}[5] = L$
$pos = Rev_{T_1}[y] = Rev_{T_1}[5] = 5$
$q =$ Anfangsposition des Buckets von $A_{T_1}[pos] = A_{T_1}[5] = 5$
$q = pos$: Suffix 5 ist an der Anfangsposition
Anfangsposition des gerade betrachteten Buckets um 1 erhöhen,
d.h., Anfangsposition ist 6
$i = 5$
$x = A_{T_1}[i] = A_{T_1}[5] = 5$
$y = x - 1 = 5 - 1 = 4$
$z = Typ_{T_1}[y] = Typ_{T_1}[4] = L$
$pos = Rev_{T_1}[y] = Rev_{T_1}[4] = 10$
$q =$ Anfangsposition des Buckets von $A_{T_1}[pos] = A_{T_1}[10] = 10$
$q = pos$: Suffix 4 ist an der Anfangsposition
Anfangsposition des gerade betrachteten Buckets um 1 erhöhen,
d.h., Anfangsposition ist 11[1]
$i = 6$
$x = A_{T_1}[i] = A_{T_1}[6] = 1$
$y = x - 1 = 1 - 1 = 0$
Suffix 0 existiert nicht
$i = 7$
$x = A_{T_1}[i] = A_{T_1}[7] = 8$
$y = x - 1 = 8 - 1 = 7$
$z = Typ_{T_1}[y] = Typ_{T_1}[7] = S$
Bei Typ-S-Suffixen ist keine Aktion nötig.
$i = 8$
$x = A_{T_1}[i] = A_{T_1}[8] = 3$
$y = x - 1 = 3 - 1 = 2$

1 Es existiert keine Position 11 im Array A_{T_1}. In einer Implementierung müsste das erkannt und verhindert werden, dass auf eine solche Position zugegriffen wird.

$z = Typ_{T_1}[y] = Typ_{T_1}[2] = S$
Bei Typ-S-Suffixen ist keine Aktion nötig.
$i = 9$
$x = A_{T_1}[i] = A_{T_1}[9] = 9$
$y = x - 1 = 9 - 1 = 8$
$z = Typ_{T_1}[y] = Typ_{T_1}[8] = S$
Bei Typ-S-Suffixen ist keine Aktion nötig.
$i = 10$
$x = A_{T_1}[i] = A_{T_1}[10] = 4$
$y = x - 1 = 4 - 1 = 3$
$z = Typ_{T_1}[y] = Typ_{T_1}[3] = S$
Bei Typ-S-Suffixen ist keine Aktion nötig.
Das fertig sortierte Array A_{T_1} und damit das Suffix Array des Textes T_1 sieht für das Beispiel wie folgt aus[1]:

	1	2	3	4	5	6	7	8	9	10
A_{T_1}	10	7	2	6	5	1	8	3	9	4
	$	a	i	l	p	s	t	t	z	z
		t	t	a	l	i	z	z	$	p
		z	z	t	a	t	$	p		l
		$	p	z	t	z		l		a
			l	$	z	p		a		t
			a		$	l		t		z
			t			a		z		$
			z			t		$		
			$			z				
						$				

Im Fall der Typ-L-Suffixe wird abschließend das Array A_{T_2} von rechts nach links durchlaufen.[2] Wird $A_{T_2}[i]$ betrachtet und handelt es sich bei $A_{T_2}[i] - 1$ um ein Typ-S-Suffix, so wird $A_{T_1}[i] - 1$ an das aktuelle Ende seines Buckets in A_{T_2} getauscht und das Ende um eins nach links verschoben.

1 Zur Veranschaulichung sind die eigentlich nicht enthaltenen Zeichenketten der Suffixe unterhalb der unteren doppelten Linie dargestellt.
2 Vgl. Ko u.a. (2003), S. 205; Ko u.a. (2005), S. 148.

Für das Beispiel sieht der Ablauf wie folgt aus:

	1	2	3	4	5	6	7	8	9	10
A_{T_2}	10	7	3	4	6	5	1	2	8	9

$i = 10$
$x = A_{T_2}[i] = A_{T_2}[10] = 9$
$y = x - 1 = 9 - 1 = 8$
$z = Typ_{T_2}[y] = Typ_{T_2}[8] = S$
$pos = Rev_{T_2}[y] = Rev_{T_2}[8] = 9$
$q =$ Endposition des Buckets von $A_{T_2}[pos] = A_{T_2}[9] = 9$
$q = pos$: Suffix 8 ist an der Endposition
Endposition des gerade betrachteten Buckets um 1 verringern,
d.h., die Endposition ist 8
$i = 9$
$x = A_{T_2}[i] = A_{T_2}[9] = 8$
$y = x - 1 = 8 - 1 = 7$
$z = Typ_{T_2}[y] = Typ_{T_2}[7] = S$
$pos = Rev_{T_2}[y] = Rev_{T_2}[7] = 2$
$q =$ Endposition des Buckets von $A_{T_2}[pos] = A_{T_2}[2] = 2$
$q = pos$: Suffix 7 ist an der Endposition
Endposition des gerade betrachteten Buckets um 1 verringern,
d.h., die Endposition ist 1
$i = 8$
$x = A_{T_2}[i] = A_{T_2}[8] = 2$
$y = x - 1 = 2 - 1 = 1$
$z = Typ_{T_2}[y] = Typ_{T_2}[1] = S$
$pos = Rev_{T_2}[y] = Rev_{T_2}[1] = 7$
$q =$ Endposition des Buckets von $A_{T_2}[pos] = A_{T_2}[7] = 7$
$q = pos$: Suffix 1 ist an der Endposition
Endposition des gerade betrachteten Buckets um 1 verringern,
d.h., die Endposition ist 6
$i = 7$
$x = A_{T_2}[i] = A_{T_2}[7] = 1$
$y = x - 1 = 1 - 1 = 0$
Suffix 0 existiert nicht
$i = 6$
$x = A_{T_2}[i] = A_{T_2}[6] = 5$

$y = x - 1 = 5 - 1 = 4$
$z = Typ_{T_2}[y] = Typ_{T_2}[4] = S$
$pos = Rev_{T_2}[y] = Rev_{T_2}[4] = 4$
$q =$ Endposition des Buckets von $A_{T_2}[pos] = A_{T_2}[4] = 4$
$q = pos$: Suffix 4 ist an der Endposition
Endposition des gerade betrachteten Buckets um 1 verringern,
d.h., die Endposition ist 3
$i = 5$
$x = A_{T_2}[i] = A_{T_2}[5] = 6$
$y = x - 1 = 6 - 1 = 5$
$z = Typ_{T_2}[y] = Typ_{T_2}[5] = L$
Bei Typ-L-Suffixen ist keine Aktion nötig.
$i = 4$
$x = A_{T_2}[i] = A_{T_2}[4] = 4$
$y = x - 1 = 4 - 1 = 3$
$z = Typ_{T_2}[y] = Typ_{T_2}[3] = S$
$pos = Rev_{T_2}[y] = Rev_{T_2}[3] = 3$
$q =$ Endposition des Buckets von $A_{T_2}[pos] = A_{T_2}[3] = 3$
$q = pos$: Suffix 3 ist an der Endposition
Endposition des gerade betrachteten Buckets um 1 verringern,
d.h., die Endposition ist 2
$i = 3$
$x = A_{T_2}[i] = A_{T_2}[3] = 3$
$y = x - 1 = 3 - 1 = 2$
$z = Typ_{T_2}[y] = Typ_{T_2}[2] = L$
Bei Typ-L-Suffixen ist keine Aktion nötig.
$i = 2$
$x = A_{T_2}[i] = A_{T_2}[2] = 7$
$y = x - 1 = 7 - 1 = 6$
$z = Typ_{T_2}[y] = Typ_{T_2}[6] = L$
Bei Typ-L-Suffixen ist keine Aktion nötig.
$i = 1$
$x = A_{T_2}[i] = A_{T_2}[1] = 10$
$y = x - 1 = 10 - 1 = 9$
$z = Typ_{T_2}[y] = Typ_{T_2}[9] = L$
Bei Typ-L-Suffixen ist keine Aktion nötig.

Das fertig sortierte Array A_{T_2} und damit das Suffix Array für Text T_2 sieht für das Beispiel wie folgt aus[1]:

	1	2	3	4	5	6	7	8	9	10
A_{T_2}	10	7	3	4	6	5	1	2	8	9
	$	a	e	h	l	p	s	t	t	z
		t	h	p	a	l	t	e	z	$
		z	p	l	t	a	e	h	$	
		$	l	a	z	t	h	p		
			a	t	$	z	p	l		
			t	z		$	l	a		
			z	$			a	t		
			$				t	z		
							z	$		
							$			

Dieser Algorithmus ist nicht der einzige, der aus einem Text T direkt ein Suffix Array erstellen kann. Weitere Algorithmen können bspw. in Sadakane (1998), Larsson u.a. (1999), Schürmann u.a. (2007), Kim u.a. (2003), Kärkkäinen u.a. (2003) und Kärkkäinen u.a. (2006) nachgelesen werden. Einen Überblick über weitere Algorithmen und eine Klassifikation der Algorithmen findet sich in Puglisi u.a. (2007).
Der Algorithmus von Ko und Aluru wurde in dieser Arbeit ausgewählt, da er zum einen leicht nachzuvollziehen und leicht zu implementieren ist und zum anderen eventuell weiter verbessert werden kann, ohne den Algorithmus komplett austauschen zu müssen. Einer der schnellsten und speichereffizientesten Algorithmen, um aus einem Text direkt ein Suffix Array zu erstellen, ist laut Puglisi u.a. (2007)[2] der Algorithmus von Maniscalco u.a. (2006). Dieser verwendet die Idee der Sortierung eines kleinen Teils der zu sortierenden Suffixe und reduziert diesen Teil noch weiter.

1 Zur Veranschaulichung sind die eigentlich nicht enthaltenen Zeichenketten der Suffixe unterhalb der unteren doppelten Linie dargestellt.
2 Vgl. Puglisi u.a. (2007), S. 25, 28.

Gepaart mit einer Strategie zur Reduzierung von Cache Fehlzugriffen[1] sowie einer Behandlung von Zeichenwiederholungen in den zu sortierenden Suffixen wird eine schnellere Sortierung der zu sortierenden Suffixe erreicht. Die nächsten Schritte, also die Sortierung der restlichen Suffixe des Textes mit Hilfe der kleinen, bereits sortierten Menge der Suffixe, entspricht der Verarbeitung im Algorithmus von Ko und Aluru.

Auch der Algorithmus von Nong u.a. (2009) beruht auf dem Algorithmus von Ko und Aluru. Hier wird die Anzahl an vorzusortierenden Suffixen weiter eingeschränkt und die Idee einer Vorsortierung eines kleinen Teils der Suffixe, welche eine „automatische“ Sortierung des Rests der Suffixe nach sich zieht, auch auf die Sortierung der vorzusortierenden Suffixe angewendet. Das bedeutet, die Algorithmen ähneln sich sehr, der Algorithmus von Nong u.a. (2009) verbessert aber den Algorithmus von Ko und Aluru weiter.[2]

Das bedeutet, die Implementierung des Algorithmus[3] lässt sich im Hinblick auf die Laufzeit noch weiter verbessern. Da dies nicht Thema der vorliegenden Arbeit ist, erfolgte keine Implementierung dieser möglichen Verbesserungen, sondern lediglich die Auswahl eines Algorithmus, mit dem dies möglich ist, um eine eventuelle zukünftige Anpassung nicht zu erschweren.

3.1.4.4 Implementierung des Algorithmus nach Ko und Aluru zur direkten Erstellung eines Suffix Arrays für einen Text

3.1.4.4.1 Datenstruktur des Suffix Arrays

In der Implementierung des Algorithmus[4] nach Ko und Aluru wird das Suffix Array zunächst als einfach verkettete Liste[5] erzeugt. Das bedeutet, jeder Knoten der ver-

1 Das Wort *Cache* (engl.) bedeutet *Versteck*. Bei einem Cache handelt es sich um einen Speicher eines Rechners, der zwar nicht so viele Daten vorhalten kann wie der Hauptspeicher, dafür aber geringere Zugriffszeiten aufweist, vgl. Böttcher (2006), S. 133. Man verwendet den Cache also zur Zwischenspeicherung von Daten und Befehlen, um die Anzahl der Zugriffe auf den Hauptspeicher zu „minimieren“ und somit die Zugriffszeit zu verkleinern, vgl. Oberschelp u.a. (2000), S. 275. Ein Fehlzugriff auf den Cache bedeutet, dass sich die Daten, auf die zugegriffen werden soll, nicht im Cache befinden und daher der langsamere Zugriff auf den Hauptspeicher erfolgen muss, vgl. Böttcher (2006), S. 133; Malz (2001), S. 107; Märtin (1994), S. 224; Oberschelp u.a. (2000), S. 275. Diese Zugriffszeit bei einem Cache Fehlzugriff ist länger im Vergleich zu einem normalen Hauptspeicher-Zugriff, vgl. Malz (2001), S. 107, und deshalb zu vermeiden.

2 Einen direkten Vergleich findet man in Nong u.a. (2009), S. 202. Aus einem Report geht sogar hervor, dass die Implementierung des Algorithmus von Nong u.a. (2009) in einer Verbesserung durch Mori (2010) ähnliche Laufzeit und eine bessere Speicherausnutzung aufweist als der Algorithmus von Maniscalco u.a. (2006), vgl. Bellet (2009), S. 10-12.

3 Siehe Kapitel 3.1.4.4.

4 Eine Erläuterung zum Software-Prototyp befindet sich in Anhang F auf S. 593.

5 Zur Erläuterung vgl. bspw. Balzert (1999), S. 600.

ketteten Liste speichert seine Daten und einen Zeiger auf seinen Nachfolger. Diese Datenstruktur wird verwendet, da das Einfügen neuer Knoten in diese Datenstruktur leicht möglich ist und das Einfügen neuer Knoten beim Erzeugen des Suffix Arrays im Vordergrund steht.
Später, nachdem alle Suffixe lexikografisch richtig nach ihrem ersten Zeichen in die Liste eingefügt wurden, wird die Datenstruktur in ein Array transformiert, in das als Elemente die Daten der Knoten gespeichert werden. Diese Datenstruktur ist die Grundlage für die weiterführende lexikografische Sortierung der Suffixe nicht nur nach ihrem ersten Zeichen, sondern nach allen Zeichen.

3.1.4.4.2 Erzeugen eines Suffix Arrays

3.1.4.4.2.1 Erzeugen des Suffix Arrays lexikografisch sortiert nach dem ersten Zeichen der Suffixe

Die Operation, die zunächst eine einfach verkettete Liste als Suffix Array erzeugt, wobei eine lexikografisch aufsteigende Sortierung nach den ersten Zeichen der in Text T auftretenden Suffixe vorgenommen wird, ist `constructArray`[1]. Innerhalb dieser Operation werden zunächst alle benötigten Variablen erzeugt und auch die Datenstrukturen für die Bucketgrenzen. Bei den Datenstrukturen für die Bucketgrenzen handelt es sich um zwei Datenstrukturen für Zeiger, die das Zeichen des Buckets und jeweils einen Zeiger auf den ersten Knoten des Buckets und den letzten Knoten des Buckets speichern. Begonnen wird mit dem ersten Zeichen des Textes. Für dieses werden die benötigten Knotendaten erzeugt und anschließend wird der Typ des Suffixes festgelegt.
Die Festlegung des Typs erfolgt in der Operation `createType`. Sie ist in Abbildung 3.3 auf S. 150 zu sehen. Innerhalb der Operation wird zeichenweise überprüft, ob das aktuelle Zeichen, in der Abbildung als „pos1“ bezeichnet, kleiner oder größer ist als das ihm nachfolgende Zeichen, in der Abbildung als „pos2“ bezeichnet. Ist das der Fall, kann der Typ des aktuellen Suffixes, im ersten Fall Typ-S, im zweiten Fall Typ-L, bestimmt und zurückgegeben werden. Sind die beiden Zeichen gleich, so wird die Operation `createType` rekursiv mit den jeweils nachfolgenden Zeichen der beiden Zeichen aufgerufen, bis die nachfolgenden Zeichen sich unterscheiden.

1 Die Schreibweise der implementierten Operationen weicht von der Schreibweise der Funktionen oder Algorithmen ab. Namen von implementierten Operationen werden wie folgt formatiert: `Operationsname`.

Gleichzeitig wird der Zähler, der die Anzahl der Typ-S-Suffixe speichert, entsprechend erhöht oder nicht.[1]

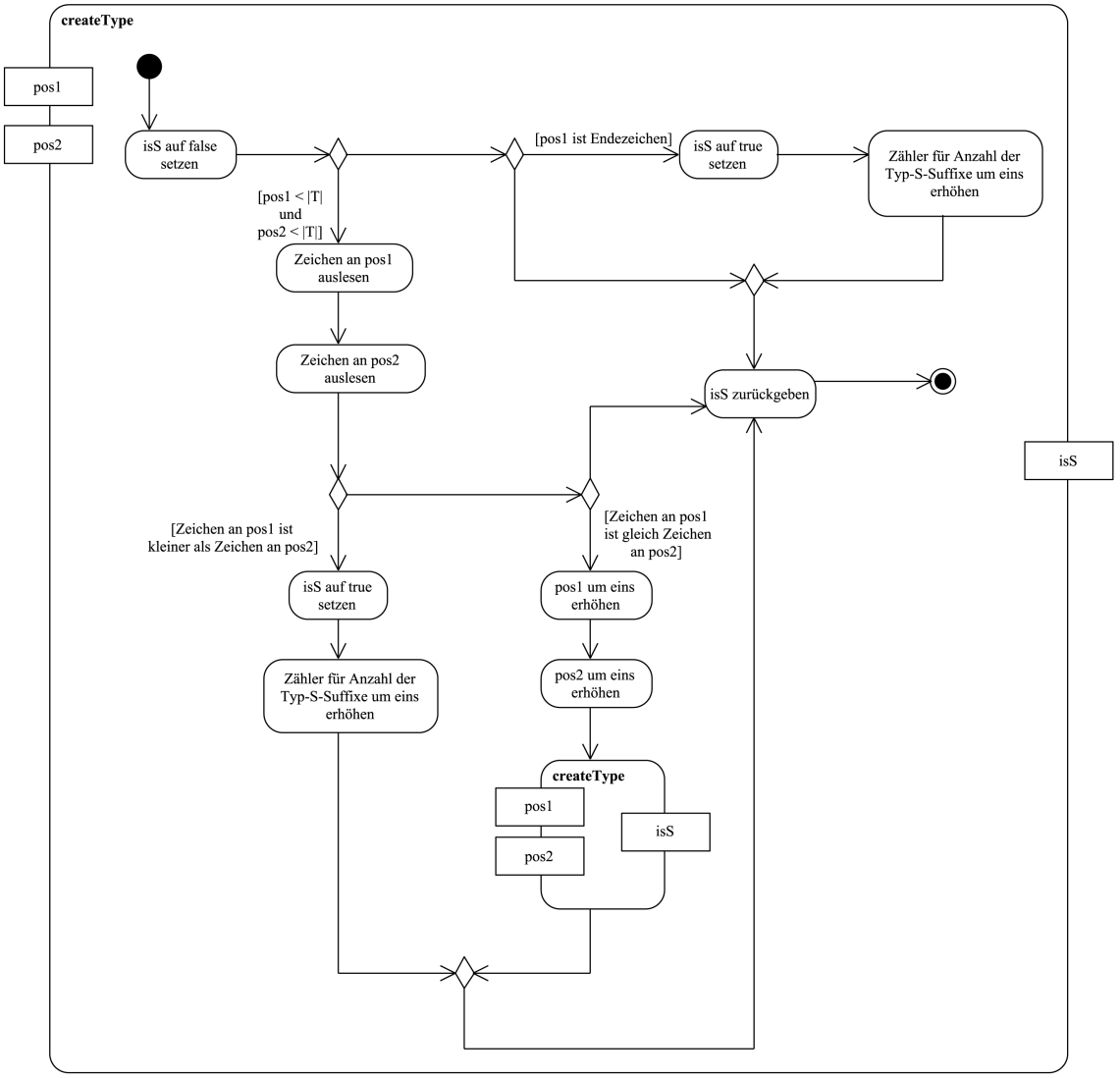


Abbildung 3.3: Ablauf der Operation `createType`

Ist der Typ des aktuellen Suffixes bestimmt, werden abhängig von diesem die entsprechenden Variablen für Distanz- und Längeninformationen aktualisiert. Die Daten des Knotens des Suffixes werden ebenfalls um die Typzuordnung ergänzt und der Knoten wird an die verkette Liste angehängt. Da es sich um das erste Suffix handelt, das eingefügt wird, erfolgt das Einfügen durch die Operation `addNode`, die in Abbildung 3.4 zu sehen ist. Diese Operation fügt den übergebenen Knoten an das

1 Es existiert kein Zähler für die Typ-L-Suffixe und demzufolge wird ihre Anzahl auch nicht erhöht. Durch die Länge der Zeichenkette und die Anzahl der Typ-S-Suffixe kann die Anzahl der Typ-L-Suffixe eindeutig bestimmt werden, siehe auch Zeile 12 in Algorithmus 7 auf S. 108.

Ende der verketteten Liste an. Da die Liste beim Einfügen des ersten Knotens noch leer ist, werden lediglich die Zeiger für den Anfang und das Ende der verketteten Liste entsprechend auf den eingefügten Knoten gesetzt.
Zusätzlich werden das erste Zeichen des gerade eingefügten Suffixes und ein Zeiger auf den gerade eingefügten Knoten in den beiden Datenstrukturen für den Anfang und das Ende des Buckets für dieses Zeichen ergänzt. Anschließend erfolgt das Hinzufügen der restlichen Suffixe der Zeichenkette. Die Abbildung 3.5 auf S. 153 zeigt den Ablauf. Das jeweils nächste Zeichen wird aus dem Text ausgelesen und ein Datenobjekt für den einzufügenden Knoten erzeugt. Dann erfolgt auch die Typbestimmung für dieses Suffix, wie bereits zuvor beschrieben, und das Datenobjekt wird ergänzt. Wie auch beim ersten Zeichen werden die entsprechenden Distanz- und Längeninformationen aktualisiert. Nun muss der Knoten an seine richtige Position innerhalb der Liste eingefügt werden. Die richtige Position bedeutet: nach dem ersten Zeichen des einzufügenden Suffixes lexikografisch sortiert.

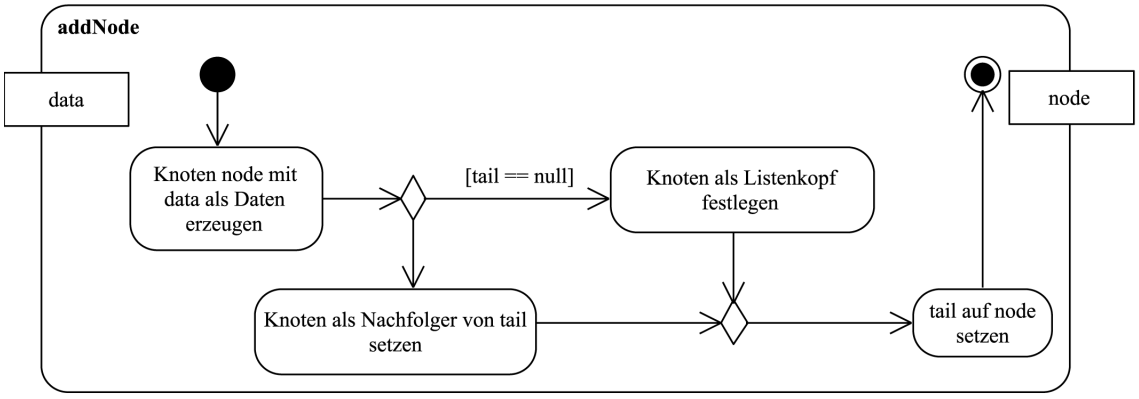


Abbildung 3.4: Ablauf der Operation `addNode`

Es können zwei Fälle unterschieden werden:

1. **Das erste Zeichen des einzufügenden Suffixes ist bereits im aktuellen Suffix Array des Textes vorhanden.**
   Wenn das erste Zeichen des einzufügenden Suffixes bereits vorhanden ist, dann wird das einzufügende Suffix am Ende des Buckets für das erste Zeichen eingefügt. Das bedeutet, es genügt, den Zeiger auf das Ende des Buckets für das Zeichen auszulesen. Dieser Knoten, auf den der Zeiger zeigt, ist der Knoten, *nach* dem der aktuelle Knoten eingefügt werden muss. Das geschieht mit der Operation `insertElementAfterNode`, die weiter unten beschrieben wird.

2. **Das erste Zeichen des einzufügenden Suffixes ist noch nicht im aktuellen Suffix Array des Textes vorhanden.**
   In diesem Fall existiert kein Zeiger auf das entsprechende Bucket, da das Zeichen noch nicht im Suffix Array vorhanden ist. Somit muss die Einfügeposition zunächst gesucht werden. Das erfolgt mit der Operation `searchForInsertPos`, die weiter unten beschrieben wird.

Tritt der erste Fall ein und wird die Operation `insertElementAfterNode` aufgerufen, so ist die Einfügeposition des hinzuzufügenden Suffix eindeutig bestimmt. Daher wird innerhalb der Operation, die in Abbildung 3.6 auf S. 154 zu sehen ist, aus den Daten ein Knoten erzeugt. Beim Einfügen selbst muss darauf geachtet werden, dass der Knoten, hinter den der Knoten eingefügt wird, einen Nachfolger haben kann. Dieser Nachfolgeknoten beschreibt ein Suffix, das mit einem anderen, lexikografisch größeren Zeichen, beginnt. Trotzdem muss diese Nachfolgebeziehung erhalten bleiben. Daher wird zunächst der Nachfolgeknoten für den Knoten, hinter dem eingefügt werden soll, bestimmt und zwischengespeichert. Existiert dieser Knoten, so werden die Nachfolgebeziehungen aktualisiert, d.h., der einzufügende Knoten wird Nachfolger des Knotens, hinter dem er eingefügt werden soll, und der ursprüngliche Nachfolger dieses Knotens wird der Nachfolger des eingefügten Knotens. Existiert kein Nachfolgerknoten, so kann es sein, dass der Knoten, hinter dem eingefügt werden soll, der letzte Knoten der Liste ist, dann wird der einzufügende Knoten ans Ende der Liste eingefügt.

Im zweiten Fall muss zunächst die richtige Einfügeposition gefunden werden. Das erfolgt in der Operation `searchForInsertPos`, die in Abbildung 3.7 auf S. 155 zu sehen ist. Diese Operation beginnt am Kopf der Liste und sucht durch Vergleiche des jeweils ersten Zeichens des aktuellen Suffixes das Suffix, das mit einem größeren Zeichen als das einzufügende Suffix beginnt. Der Vorgänger dieses Knotens, den man sich während des Durchlaufs jeweils merkt, wird als der Knoten zurückgegeben, hinter den der aktuell einzufügende Knoten eingefügt werden soll. Wird kein solcher Knoten gefunden, so bedeutet das, dass der erste Knoten der Liste ein Suffix beschreibt, das bereits mit einem größeren Zeichen als das einzufügende beginnt. Somit muss der einzufügende Knoten am Kopf der Liste eingefügt werden.

Wird dagegen ein Knoten zurückgeliefert, nach dem der einzufügende Knoten eingefügt werden soll, so wird die Operation `insertElementAfterNode`, die weiter oben beschrieben wurde, aufgerufen und fügt den Knoten wie beschrieben ein. Abschließend müssen in diesem Fall die Zeiger für den Bucketanfang und das Bucketende um das neu eingefügte Anfangszeichen ergänzt werden.

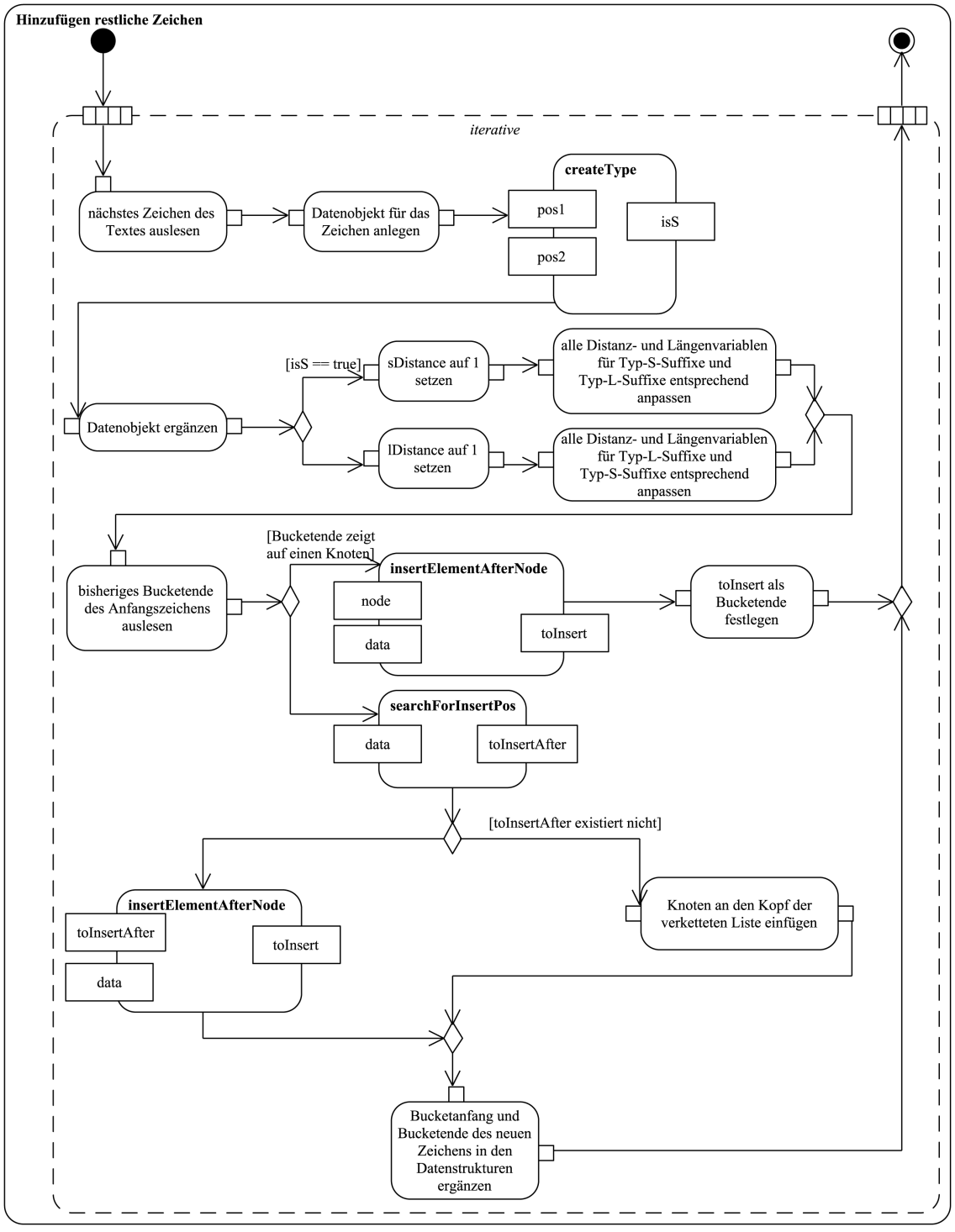


Abbildung 3.5: Hinzufügen der restlichen Zeichen zur ersten Version des Suffix Arrays

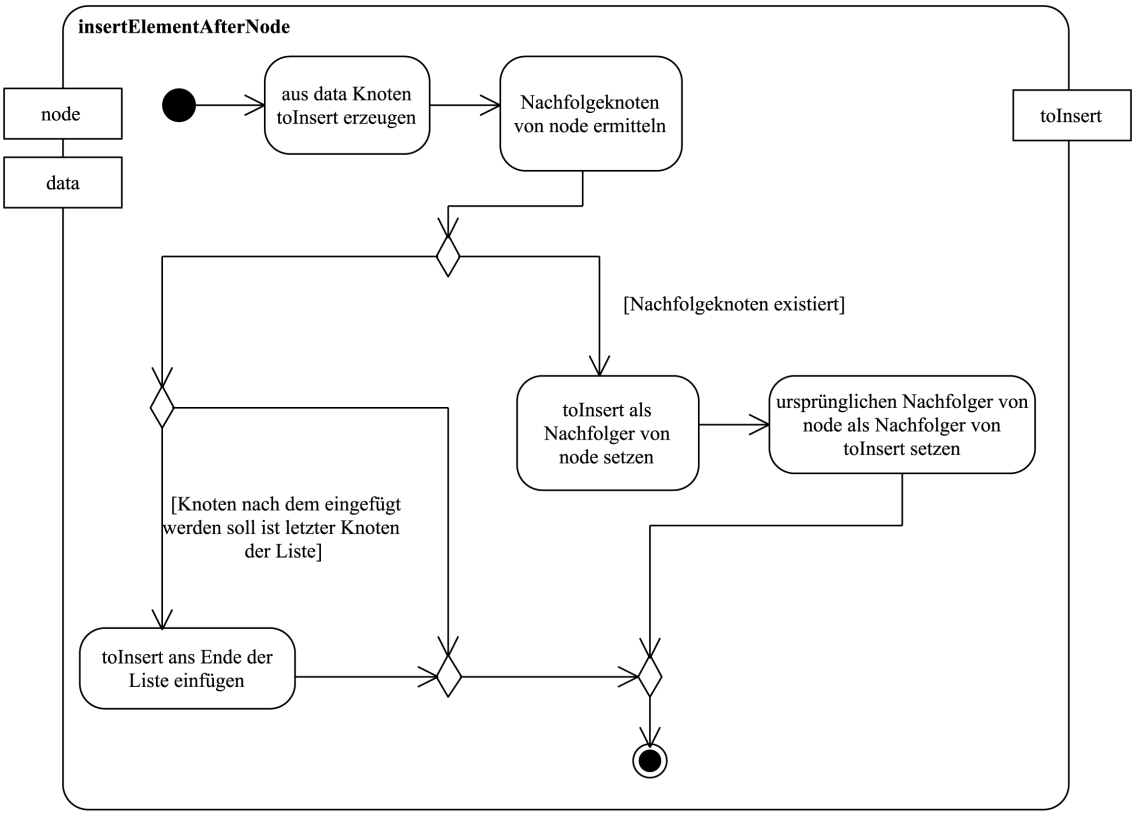


Abbildung 3.6: Ablauf der Operation `insertElementAfterNode`

Sind auf diese Weise alle Zeichen des Textes hinzugefügt worden, ist die verkettete Liste das Suffix Array für diesen Text sortiert nach dem ersten Zeichen der Suffixe des Textes.

Der nächste Schritt ist, das Suffix Array weiter zu sortieren. Dafür wird die verkettete Liste in ein Array umgewandelt, damit die Operationen des Algorithmus von Ko und Aluru ausgeführt werden können. Die Operation `createPosAndArray` nimmt diese Umwandlung vor und erzeugt auch das Array *Rev*. Innerhalb der Operation, die in Abbildung 3.8 auf S. 155 zu sehen ist, wird die verkettete Liste von vorne nach hinten durchlaufen und die Knoteninhalte werden jeweils an der nächsten Position des Arrays, welches jetzt das Suffix Array darstellt, gespeichert. Gleichzeitig werden im Array *Rev* an der Position der Suffixe im Text die Positionen der Suffixe im Suffix Array gespeichert.

Ist das Suffix Array vollständig umgewandelt, so ist auch die Operation `construct-Array`, die in Abbildung 3.9 auf S. 156 zu sehen ist, abgeschlossen und das Suffix Array in einer ersten Version erzeugt. Das bedeutet, die ersten beiden Schritte des Algorithmus nach Ko und Aluru sind abgearbeitet.

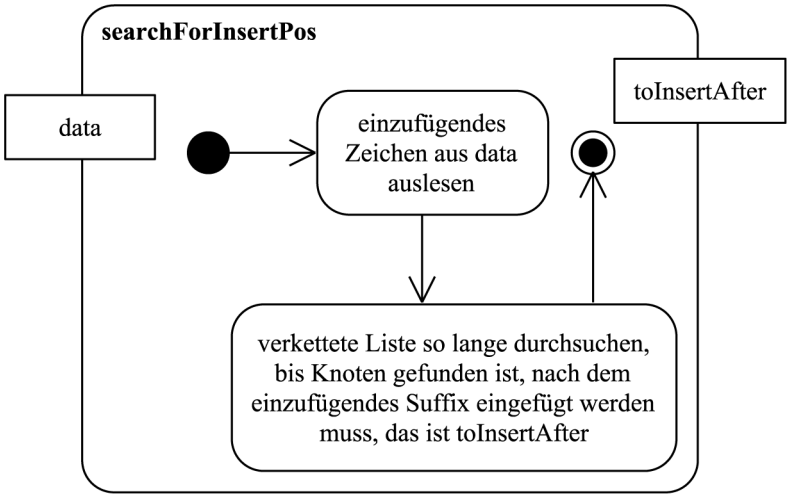


Abbildung 3.7: Ablauf der Operation `searchForInsertPos`

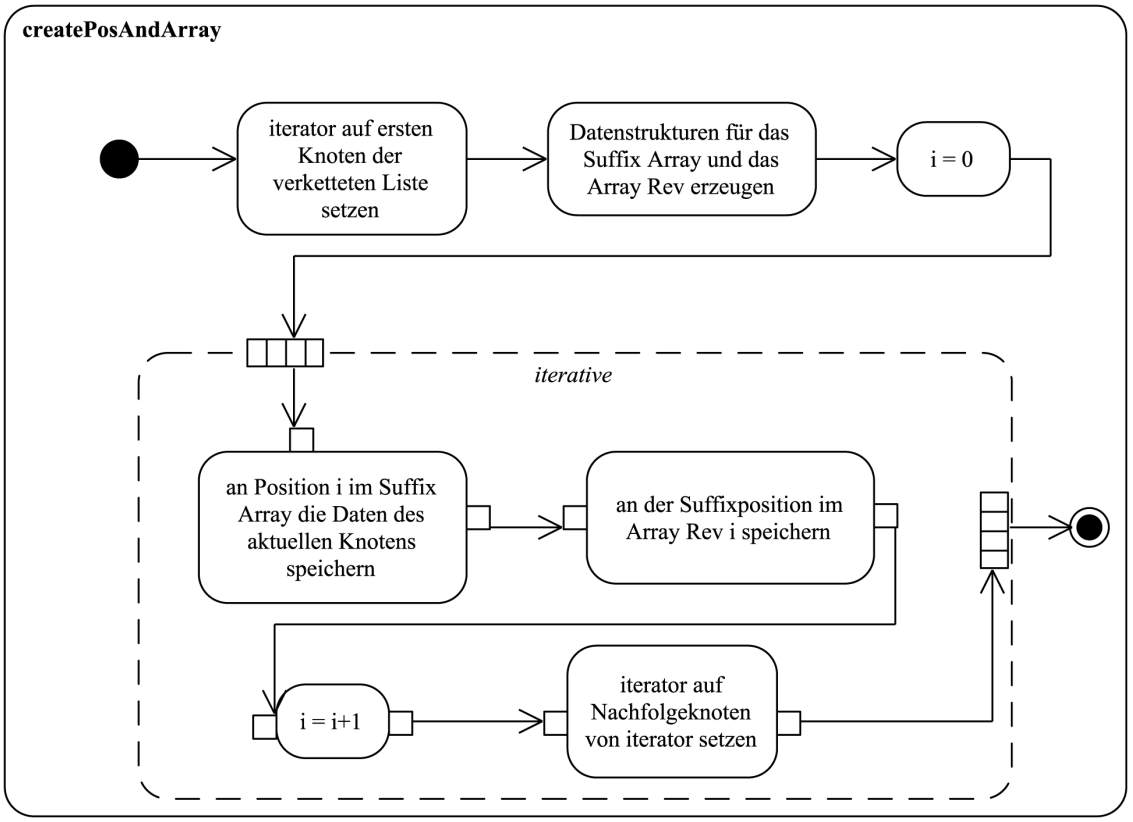


Abbildung 3.8: Ablauf der Operation `createPosAndArray`

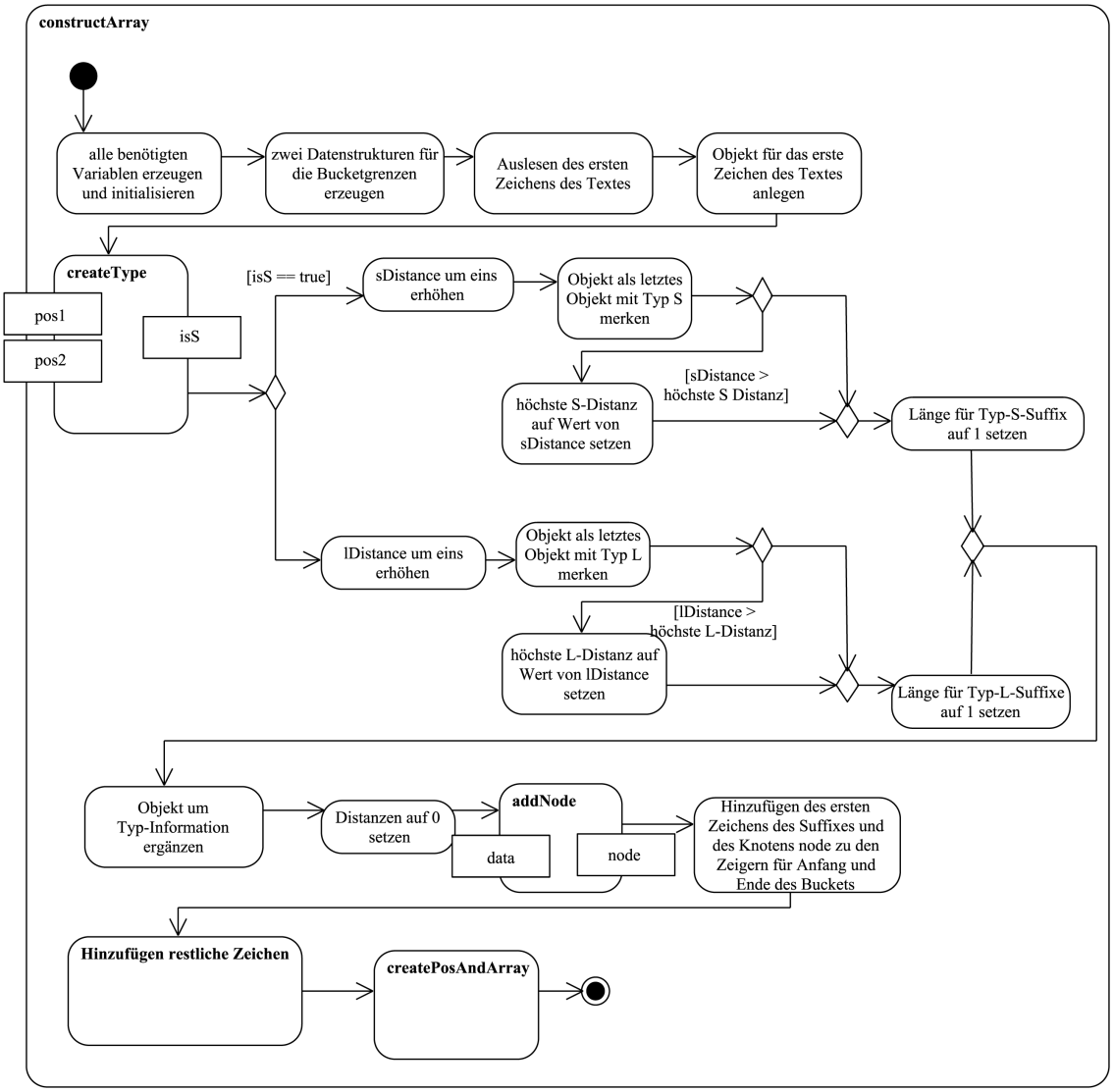


Abbildung 3.9: Ablauf der Operation `constructArray`

3.1.4.4.2.2 Erzeugen des lexikografisch nach allen Zeichen sortierten Suffix Arrays

Nachdem das Suffix Array in seiner vorläufigen Form - also nach dem ersten Zeichen der Suffixe des Textes lexikografisch aufsteigend sortiert - vorhanden ist, erfolgt die restliche Sortierung der Suffixe in der Operation `sortArray`. Diese Operation ist in Abbildung 3.10 auf S. 157 zu sehen. Innerhalb der Operation werden drei weitere Operationen aufgerufen. Die ersten beiden sind gleich, egal ob Typ-S-Suffixe oder Typ-L-Suffixe vorsortiert werden. Die letzte Operation unterscheidet sich dann je nachdem, welche Typen vorsortiert wurden.

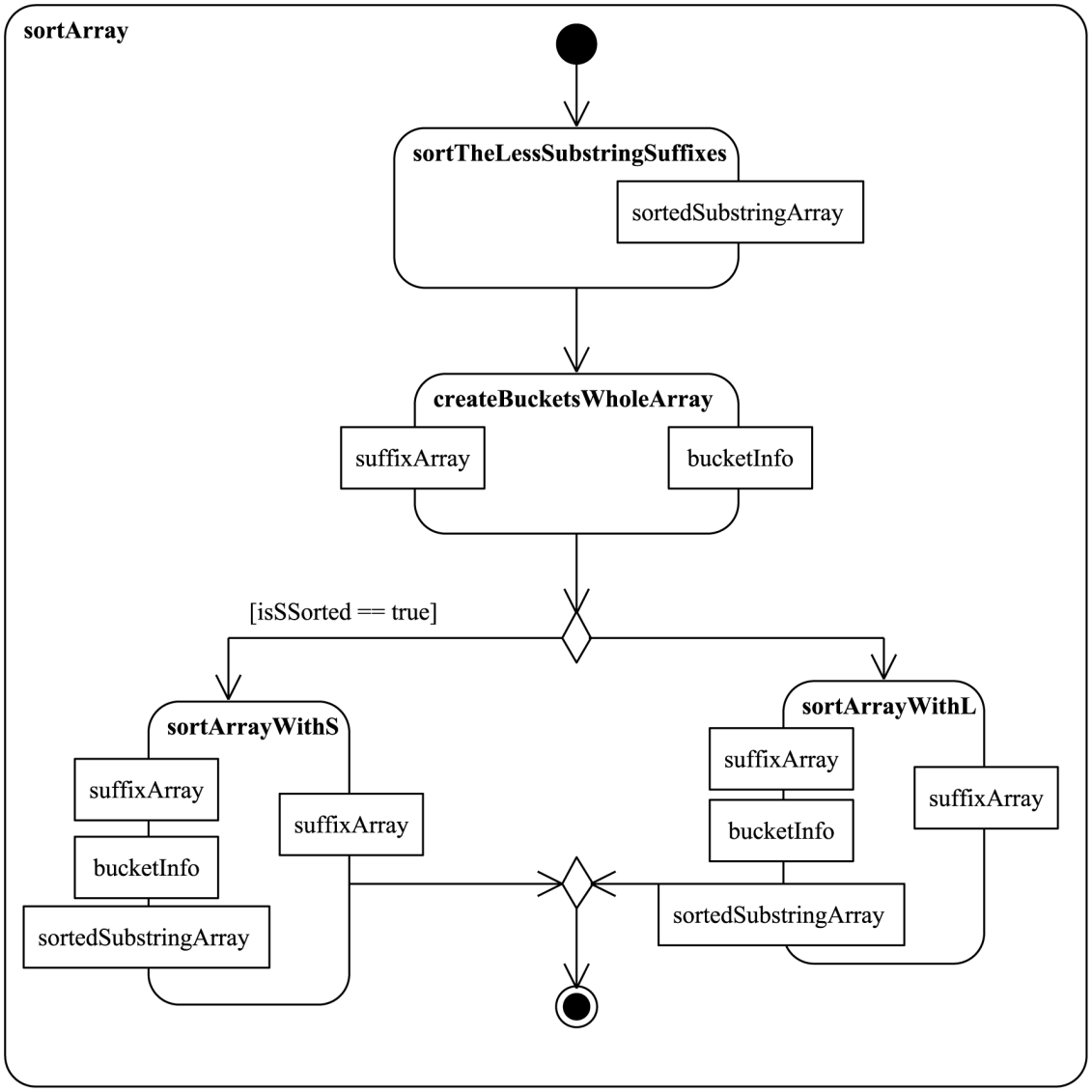


Abbildung 3.10: Ablauf der Operation `sortArray`

Die erste aufgerufene Operation trägt den Namen `sortTheLessSubstringSuffixes`. Zu sehen ist sie in Abbildung 3.11 auf S. 159. In der Operation wird zunächst eine Datenstruktur für die vorzusortierenden Suffixe angelegt und die Anzahl der Typ-L-Suffixe im Text berechnet. Die Berechnung erfolgt über die Subtraktion der Anzahl der Typ-S-Suffixe, die bei der Bestimmung des Typs der Suffixe mitgezählt wurde, von der Textlänge.[1] Ist die Anzahl der Typ-L-Suffixe kleiner als die Anzahl der Typ-S-Suffixe, so werden die beiden Operationen `createLists` und `sortSubstrings` mit

1 Im Gegensatz zu dem Algorithmus von Ko und Aluru legt die Verfasserin fest, dass das eindeutige Endezeichen immer Typ-S ist und nicht wie im Algorithmus beide Typen erhält. Dadurch ist jedes Suffix eindeutig einem Typ zugeordnet und die Berechnung der jeweiligen Anzahl kann auch eindeutig erfolgen.

dem Parameter `false` aufgerufen und die Variable `isSSorted` wird auf `false` gesetzt. Im anderen Fall - die Anzahl der Typ-L-Suffixe ist größer oder gleich der Anzahl der Typ-S-Suffixe - werden beide Operationen mit dem Parameter `true` aufgerufen und die Variable `isSSorted` wird ebenfalls auf `true` gesetzt. Abschließend gibt die Operation das Array C zurück, also das Array mit den sortierten Suffixen des vorzusortierenden Typs. In der Implementierung heißt dieses Array `sortedSubstrings`.

Die Operation `createLists` entspricht dem dritten Schritt des Algorithmus von Ko und Aluru. Es erfolgt also die Konstruktion von m Listen, wobei m die höchste S- oder L-Distanz meint. Durch den Aufruf mit dem Parameter `true` für Typ-S-Suffixe und `false` für Typ-L-Suffixe wird die Information weitergegeben, welche Distanz für das Erstellen der Listen ausschlaggebend ist. Je nachdem, welche Suffixtypen vorsortiert werden sollen, wird eine Liste, in der Implementierung ein Vektor, in der Länge der höchsten Distanz entweder von Typ-L-Suffixen oder von Typ-S-Suffixen erzeugt.

Die Länge der höchsten Distanz ist die Anzahl der benötigten Listen, also m. Dargestellt werden die m Listen aus dem Algorithmus als Liste mit m Elementen, wobei die Elemente wiederum selbst Listen sind, da hier die Textpositionen der einzelnen Suffixe gespeichert werden sollen. Das bedeutet, pro Element dieses *äußeren* Vektors wird ein weiterer *innerer* Vektor erzeugt, um in diesem die Textpositionen speichern zu können. Ein weiterer Vorteil dieser Art des Speicherns ist, dass jederzeit auf die Liste mit der gewünschten Distanz direkt zugegriffen werden kann.

Nachdem die Erzeugung dieser Datenstrukturen abgeschlossen ist, wird das Suffix Array Element für Element durchlaufen. Solange Elemente existieren, wird eine Variable `distance` auf einen negativen Wert gesetzt.[1] Danach wird die entsprechende Distanz - L oder S - für das aktuelle Element ausgelesen. Ist diese größer Null, so soll die Textposition des aktuellen Elements im inneren Vektor der gerade ausgelesenen Distanz gespeichert werden. D.h., zunächst wird aus dem äußeren Vektor der innere, der an der Position der aktuellen Distanz gespeichert wurde, ausgelesen. Danach wird die Textposition des aktuellen Elements des Suffix Arrays an diesen inneren Vektor angehängt. Ist die ausgelesene Distanz nicht größer Null, so wird dieser Schritt übersprungen, da die Textposition dann nicht in den Listen gespeichert wird. Abschließend wird das nächste Element des Suffix Arrays ausgelesen. Falls es existiert, wird die Verarbeitung für dieses Element genau so durchgeführt. Ansonsten wird die Schleife beendet. Zum Abschluss der Operation erfolgt noch

1 Zur Erinnerung: Es werden nur Listen von Distanz 1 bis einschließlich m gebildet. Das bedeutet, Distanzen kleiner 1 werden nicht berücksichtigt.

die Bestimmung der Listenbuckets. Im Gegensatz zum Algorithmus wird dies direkt innerhalb der Operation zur Erzeugung der Listen durchgeführt. Dafür werden alle Listen durchlaufen und jeweils das erste Zeichen zweier nebeneinanderstehender Suffixe miteinander verglichen. Sind sie nicht gleich, so beginnt ab dem weiter rechts stehenden Suffix ein neues Bucket der Liste. Zu sehen ist der Ablauf der Operation `createLists` in Abbildung 3.12 auf S. 160.

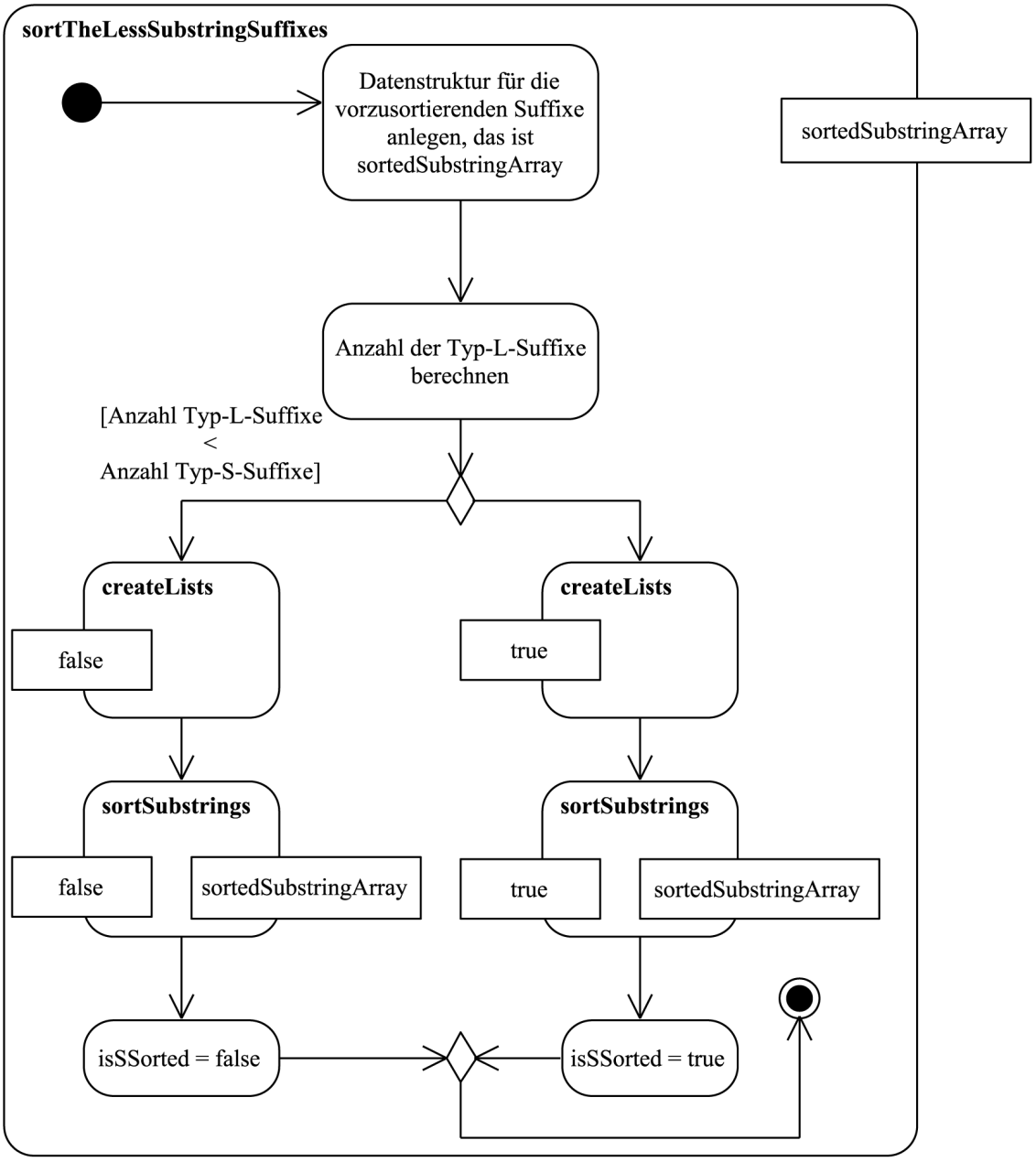


Abbildung 3.11: Ablauf der Operation `sortTheLessSubstringSuffixes`

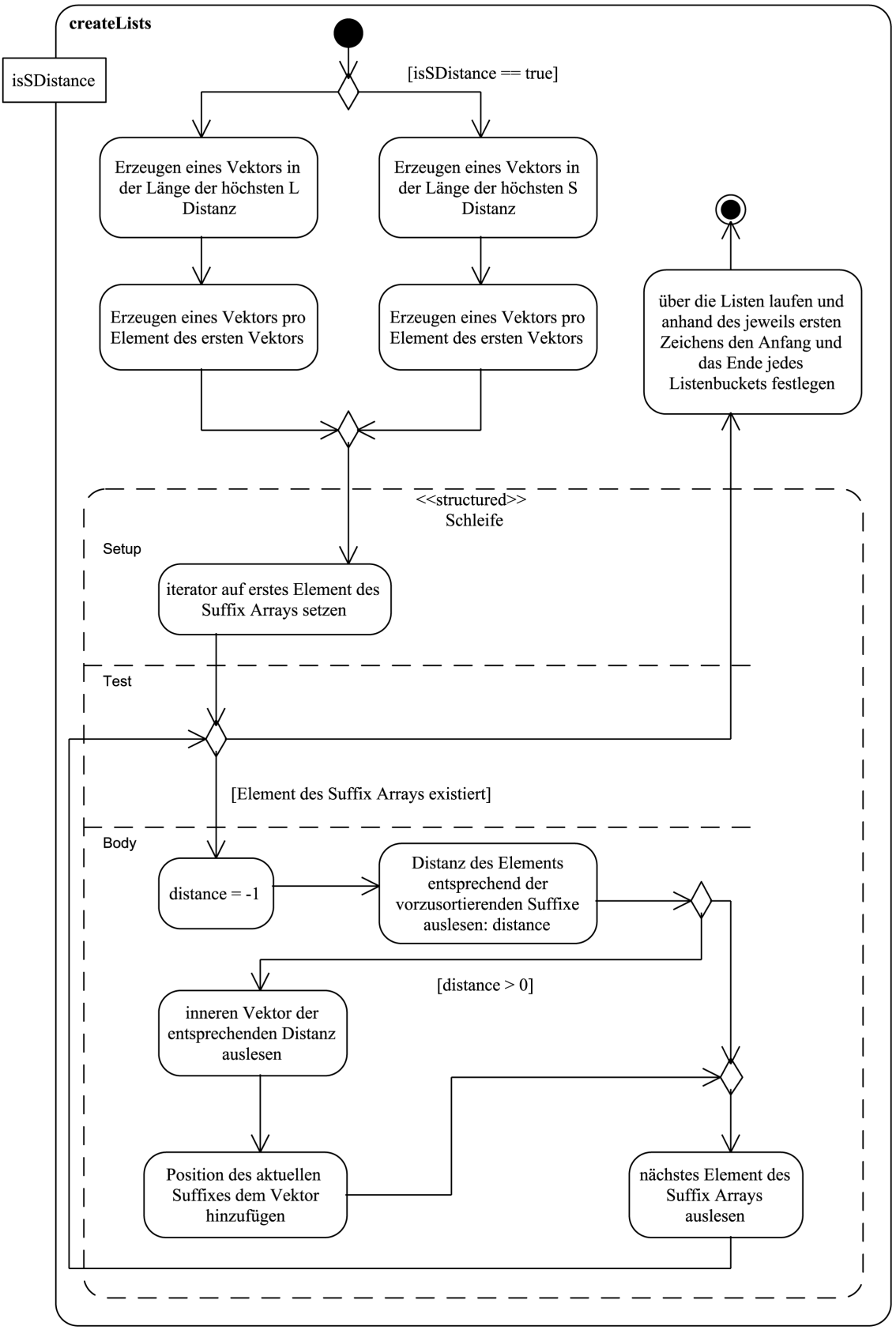


Abbildung 3.12: Ablauf der Operation `createLists`

Ist die Konstruktion der Listen abgeschlossen, müssen die vorzusortierenden Suffixe sortiert werden. Das erfolgt mit der Operation `sortSubstrings`. Der Ablauf ist in Abbildung 3.13 auf S. 162 zu sehen. Als Erstes wird eine Datenstruktur angelegt, die im Algorithmus mit Array C bezeichnet wurde. Diese heißt in der Implementierung `substringArrayToSort`. Hier werden also die Suffixe gespeichert, die vorsortiert werden sollen, um eine Sortierung des gesamten Suffix Arrays zu ermöglichen. Ein Schritt auf diesem Weg ist das Erzeugen der Arrays R und *lptr*. Dafür müssen zunächst Buckets aufgrund des ersten Zeichens des betrachteten Suffixes in Array C vorhanden sein. Daher wird die Operation `createBuckets` aufgerufen. Sie ist in Abbildung 3.15 auf S. 164 zu sehen.

Existiert bereits ein Array mit Bucketinformationen über das Array C, so erfolgen keine weiteren Verarbeitungsschritte innerhalb der Operation `createBuckets`. Ist das jedoch nicht der Fall, wird das erste Element des Arrays C ausgelesen, um als Vergleichselement für das nachfolgende Element dienen zu können. Das erste Element des Arrays C ist zugleich auch Anfang des ersten Buckets, da links davon keine Elemente vorhanden sein können. Solange keine Informationen über das nächste Element vorhanden sind, bildet das erste Element des Arrays C auch das Ende seines Buckets. Diese Informationen werden in einem weiteren Array gespeichert. Innerhalb einer Schleife wird über die verbleibenden Elemente in Array C iteriert.

Beim Schleifendurchlauf wird zunächst festgelegt, wie viele Zeichen der Suffixe miteinander verglichen werden sollen.[1] Anschließend erfolgt der Vergleich des Vergleichselements mit dem aktuell betrachteten Element, begrenzt auf diese Anzahl an Zeichen.

Stimmen die beiden Zeichen nicht überein, so wird ein neues Bucket für das aktuelle Element angelegt und die Anfangs- und Endpositionen des aktuellen Buckets werden anhand der Position des Suffixes in Array C festgelegt. Tritt dagegen der Fall ein, dass die beiden Zeichen übereinstimmen, so muss kein neues Bucket erzeugt werden, sondern stattdessen das in der vorherigen Iteration erzeugte Bucket so geändert werden, dass die Endposition die Position des aktuell betrachteten Elements ist. Abschließend wird in beiden Fällen das aktuelle Element zum nächsten Vergleichselement und das nächste Element des Arrays C wird ausgelesen.

Ist die Schleife vollständig durchlaufen, erfolgt die Erzeugung eines `boolean` Arrays, das angibt, welche der Buckets von Array C noch unsortiert sind. Nachdem die Buckets gerade erst erzeugt wurden, sind sie alle noch unsortiert. Abschließend wird das Array mit den Bucketinformationen zurückgegeben.

1 Anmerkung: Je nachdem welche der m Listen durchlaufen wird, wird eine größere Anzahl an Zeichen verglichen. In diesem Fall erfolgt eine Festlegung der Buckets nach dem ersten Zeichen.

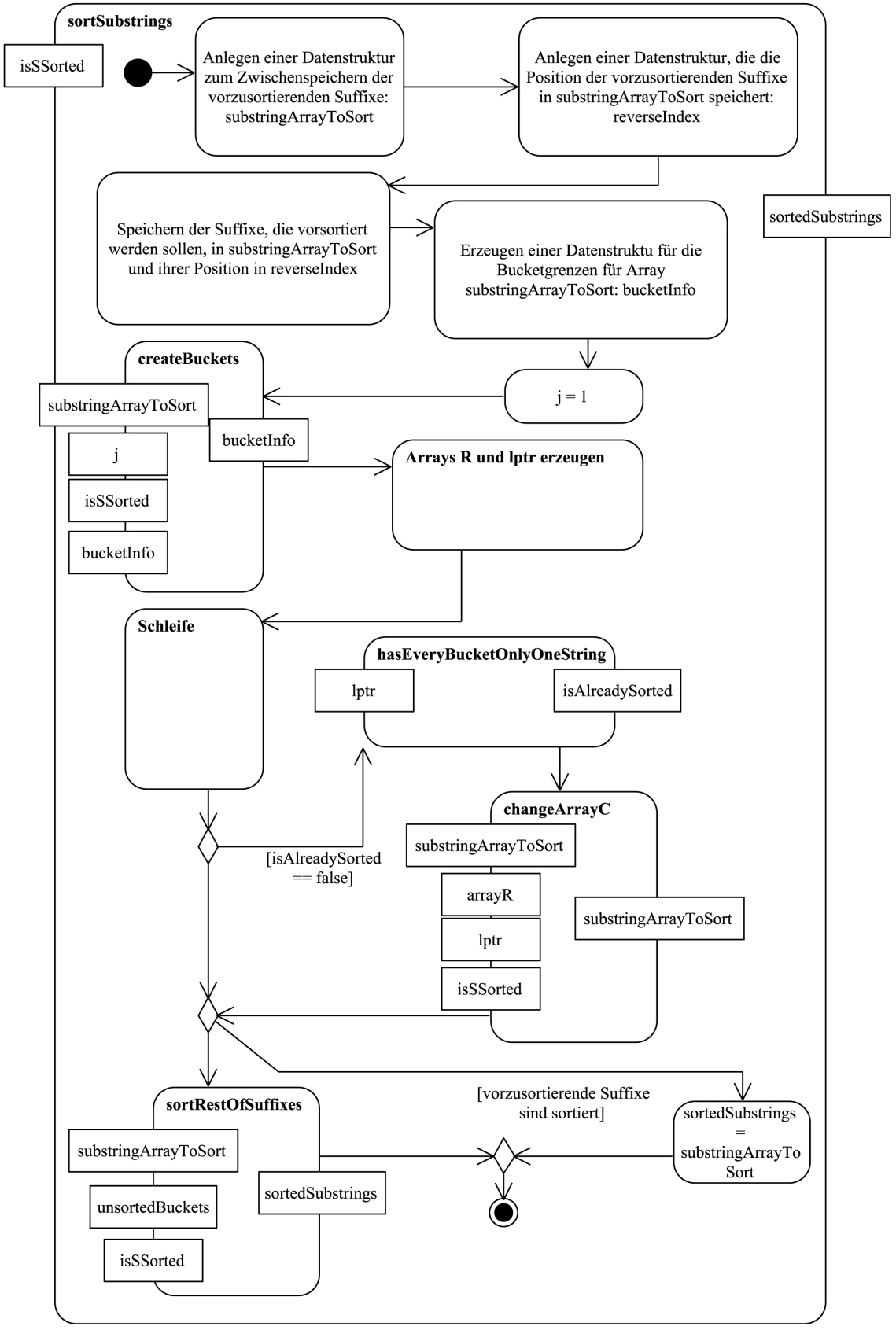


Abbildung 3.13: Ablauf der Operation `sortSubstrings`

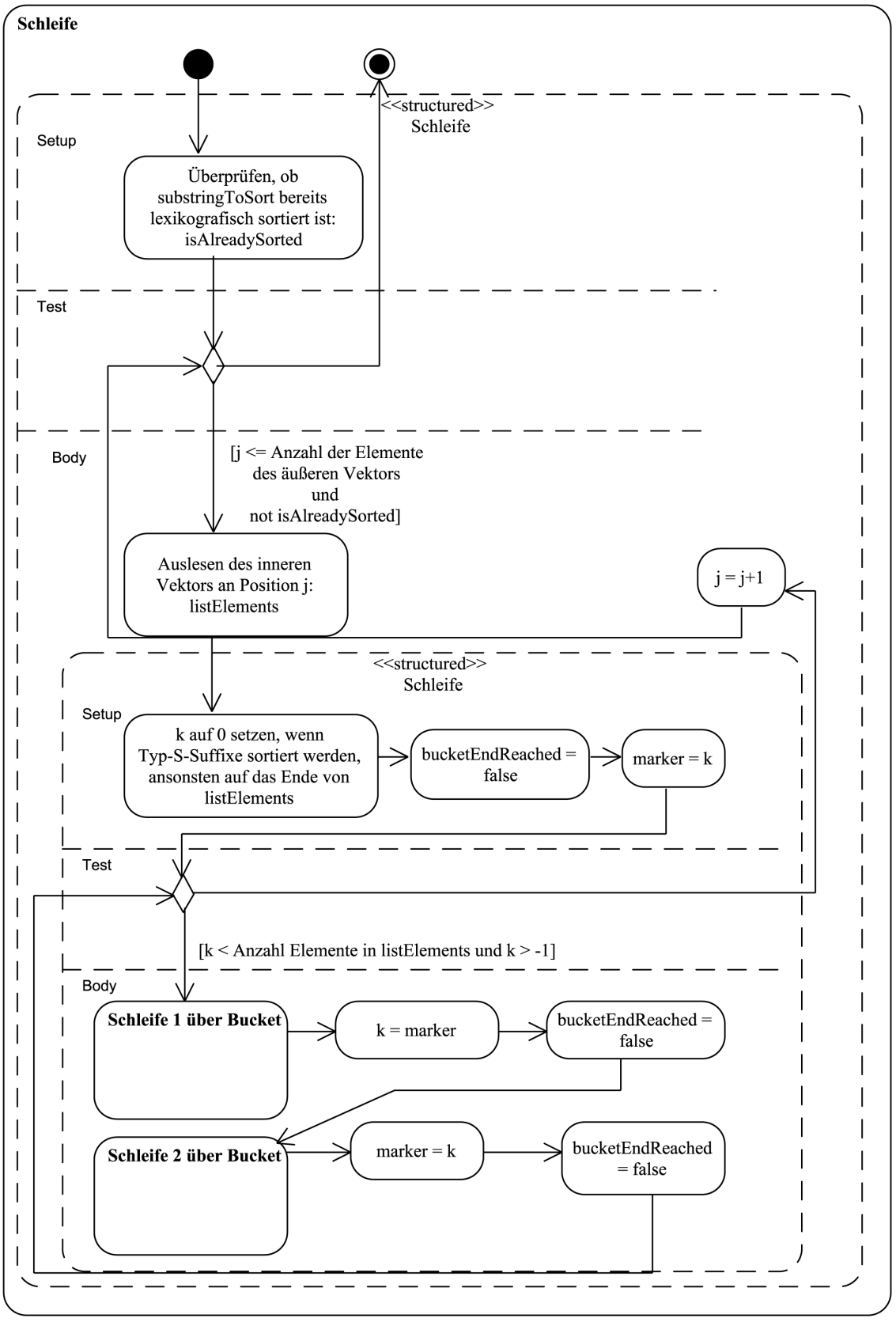


Abbildung 3.14: Ablauf der Schleife innerhalb der Operation `sortSubstrings`

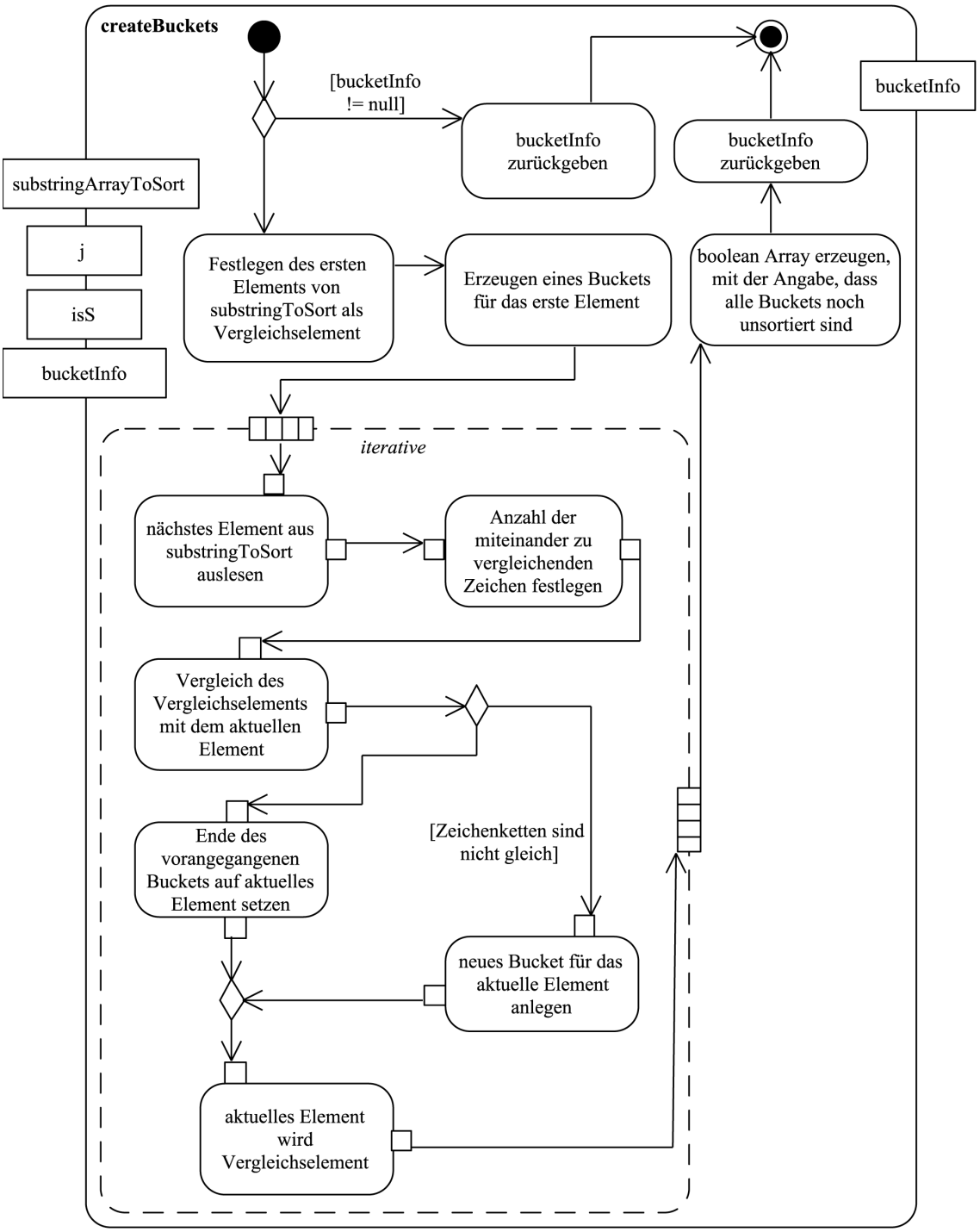


Abbildung 3.15: Ablauf der Operation `createBuckets`

Nachdem die Buckets für Array C erzeugt sind, können die Arrays R und $lptr$ angelegt werden. Beide werden mit -1 initialisiert, d.h., jedes Element der beiden Arrays erhält als Anfangswert eine -1. Wichtig ist dies später bei der Sortierung, da

eine -1 in Array R angibt, dass sich die entsprechende Textposition nicht in Array C befindet, und eine -1 in Array *lptr* angibt, dass die Suffixe im entsprechenden Bucket noch nicht lexikografisch aufsteigend sortiert sind. Um die Werte für die Elemente in den beiden Arrays bestimmen zu können, muss zunächst unterschieden werden, nach welchem Suffixtyp vorsortiert wird.

- **Typ-S-Suffixe werden vorsortiert.**
  Im ersten Fall ist es so, dass in Array R die Bucketenden und in Array *lptr* die Bucketanfänge gespeichert werden. Array C enthält die *Positionen der Suffixe im Text*. In Array R sind die *Bucketenden* im Index, der der Suffixposition entspricht, gespeichert. Array *lptr* enthält die *Bucketanfänge* gespeichert im Index, der den Bucketenden entspricht. Also muss Array C von hinten nach vorne durchlaufen werden, um die Suffixposition zu ermitteln und so das Bucketende und dann erst den Bucketanfang auslesen zu können. Die ermittelten Bucketanfänge und -enden werden entsprechend in R und *lptr* gespeichert.

- **Typ-L-Suffixe werden vorsortiert.**
  Im zweiten Fall werden in Array R die Bucketanfänge und in Array *lptr* die Bucketenden gespeichert. Aus der oben erläuterten Beziehung ergibt sich, dass daher das Array C von vorne nach hinten durchlaufen werden muss, um die benötigten Positionen ermitteln und speichern zu können.

Der Ablauf ist in Abbildung 3.16 zu sehen.

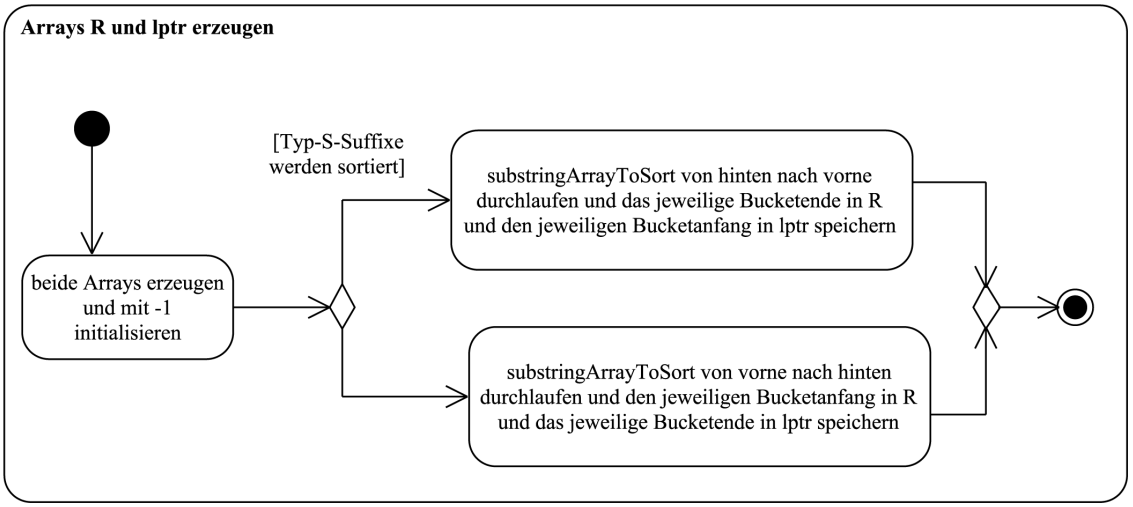


Abbildung 3.16: Erzeugen der Arrays R und *lptr*

Nachdem die beiden Arrays erzeugt wurden, wird überprüft, ob die Suffixe in Array C bereits lexikografisch aufsteigend sortiert sind. Das lässt sich am Array *lptr* ablesen: Wenn alle dort enthaltenen Werte ungleich -1 und unterschiedlich sind, so befindet sich jedes Suffix aus C in einem eigenen Bucket. Durch die Vorbedingung, dass die Suffixe zuvor nach ihrem ersten Zeichen aufsteigend lexikografisch sortiert sind, heißt das, dass sie in Array C auch nach ihrem ersten Zeichen aufsteigend lexikografisch sortiert sind. Wenn sie sich zusätzlich noch in einem eigenen Bucket befinden, so sind sie auch im Ganzen betrachtet lexikografisch aufsteigend sortiert. Tritt der Fall ein, dass schon nach der Erzeugung der Buckets auf Array C und der Erzeugung der Arrays R und *lptr* klar ist, dass die Suffixe korrekt sortiert sind, so braucht keine weitere Verarbeitung durchgeführt zu werden. Stattdessen wird das Array C so, wie es ist, zurückgegeben.

Im anderen Fall müssen diese Suffixe, wie im Algorithmus beschrieben, sortiert werden. Dazu iteriert man über alle vorher erzeugten Listen. Wird die entsprechende Schleife ausgeführt - sind also noch innere Listen nicht durchlaufen worden und ist Array C noch nicht sortiert -, so wird zunächst die erste innere Liste ausgelesen. Das ist `listElements`. Diese Liste muss nun bucketweise durchlaufen werden. Dafür wird eine Variable `k` benötigt. Da es einen Unterschied macht, ob Typ-S-Suffixe oder Typ-L-Suffixe sortiert werden, wird `k` dementsprechend gesetzt: Bei Typ S auf den Anfang der inneren Liste, bei Typ L auf das Ende derselben. Zusätzlich wird eine `boolean` Variable benötigt, die anzeigt, ob das Ende des gerade betrachteten Buckets bereits erreicht ist. Diese wird auf `false` gesetzt. Des Weiteren merkt man sich mit Hilfe einer weiteren Variablen den Wert von `k`, da laut Algorithmus das Bucket zweimal durchlaufen werden muss. Die Iteration erfolgt jedoch nur unter der Bedingung, dass der Zugriff auf die Elemente der inneren Liste auch erfolgen kann. Dazu muss `k` einen Wert haben, der nicht außerhalb der Positionen der Listenelemente liegt.

Ist das nicht der Fall, so wird sofort die nächste Liste betrachtet, sofern noch eine vorhanden ist. Ansonsten wird das erste Mal über das Bucket iteriert. Zu sehen ist der Ablauf in Abbildung 3.17 auf S. 168. Ist das Bucketende noch nicht erreicht, so wird das in der Liste enthaltene Element an Position `k` ausgelesen.[1] Werden Typ-S-Suffixe sortiert, so muss das Bucketende für das Bucket des Elements in der Liste ausgelesen werden, da das Bucket von vorne nach hinten durchlaufen wird. Im anderen Fall ist die Grenze des Buckets der Bucketanfang des Listenbuckets. Anschließend muss das zu betrachtende Suffix ermittelt werden, das ist das ausgelesene Element der Liste verringert um den Index der gerade betrachteten Liste. Für dieses Suffix wird

1 Das ist also entweder das erste Element des aktuellen Buckets oder das letzte.

nun aus Array R das zugehörige Bucket, also das Bucketende in Array C respektive der Bucketanfang in Array C, ausgelesen. Durch das Bucket lässt sich nun aus *lptr* der zugehörige aktuelle Bucketanfang oder das aktuelle Bucketende des Suffixes in Array C auslesen. Das ist die aktuelle Bucketgrenze, in der Implementierung mit `actualBoundary` bezeichnet. Sie wird verschoben, wie es der Algorithmus vorsieht. D.h., für Typ-S-Suffixe wird der Anfang um eins erhöht, für Typ-L-Suffixe das Ende um eins verringert. Anschließend wird `k` eine Position weitergesetzt, um das nächste Element, also rechts oder links vom aktuellen Element, zu betrachten. Bevor das jedoch geschieht, wird überprüft, ob die Bucketgrenze bereits erreicht wurde. Ist das der Fall, so wird `bucketEndReached` auf `true` gesetzt und demnach die Schleife abgebrochen, ansonsten wiederholt sich die Verarbeitung für das nächste Element der betrachteten Liste.

Nach dem ersten Durchlauf des Listenbuckets muss es noch einmal in genau der gleichen Richtung durchlaufen werden, um die enthaltenen Suffixe in ein neues Bucket zu überführen. Dafür wird `k` auf den Wert gesetzt, den die Variable vor dem Durchlauf der ersten Schleife über das Bucket hatte, und `bucketEndReached` wieder auf `false`. Danach erfolgt der zweite Durchlauf über das Listenbucket. Er ist in Abbildung 3.18 auf S. 169 zu sehen.

Dieser Durchlauf entspricht zu Beginn dem des ersten Durchlaufs durch das Listenbucket, d.h., es werden die Werte der gleichen Variablen ausgelesen. Nach der zweiten Unterscheidung, ob es sich um Typ-S- oder Typ-L-Suffixe handelt, erfolgt jedoch je nachdem eine andere Verarbeitung. So wird, wie im Algorithmus vorgesehen, ein neues Bucketende für Typ-S-Suffixe gesetzt und der dazugehörige Bucketanfang ausgelesen. Hatte dieses neue Bucket noch keinen Anfang, ist also der Wert in *lptr* -1, so wird das Bucketende als Bucketanfang gewählt. War jedoch bereits ein Bucketanfang vorhanden, so wird er um eins verringert und gespeichert. Für ein Typ-L-Suffix erfolgt dies analog, nur dass Anfang und Ende vertauscht sind. Abschließend wird in beiden Fällen wieder `k` verrückt und die `boolean` Variable für das Erreichen des Bucketendes entsprechend angepasst.

Nachdem auch diese Schleife komplett durchlaufen ist, wird der Wert von `k` wieder zwischengespeichert, damit auch das nächste Bucket zweimal durchlaufen werden kann, und die Angabe, dass das Bucketende erreicht wurde, auf `false` gesetzt. Dann wird die Schleifenbedingung geprüft, ob die innere Liste noch weitere Elemente enthält, und dann gegebenenfalls die ganze Verarbeitung für die nächsten Listenbuckets wiederholt. Sollte die gerade betrachtete Liste keine Elemente mehr haben, so wird die nächste Liste betrachtet.

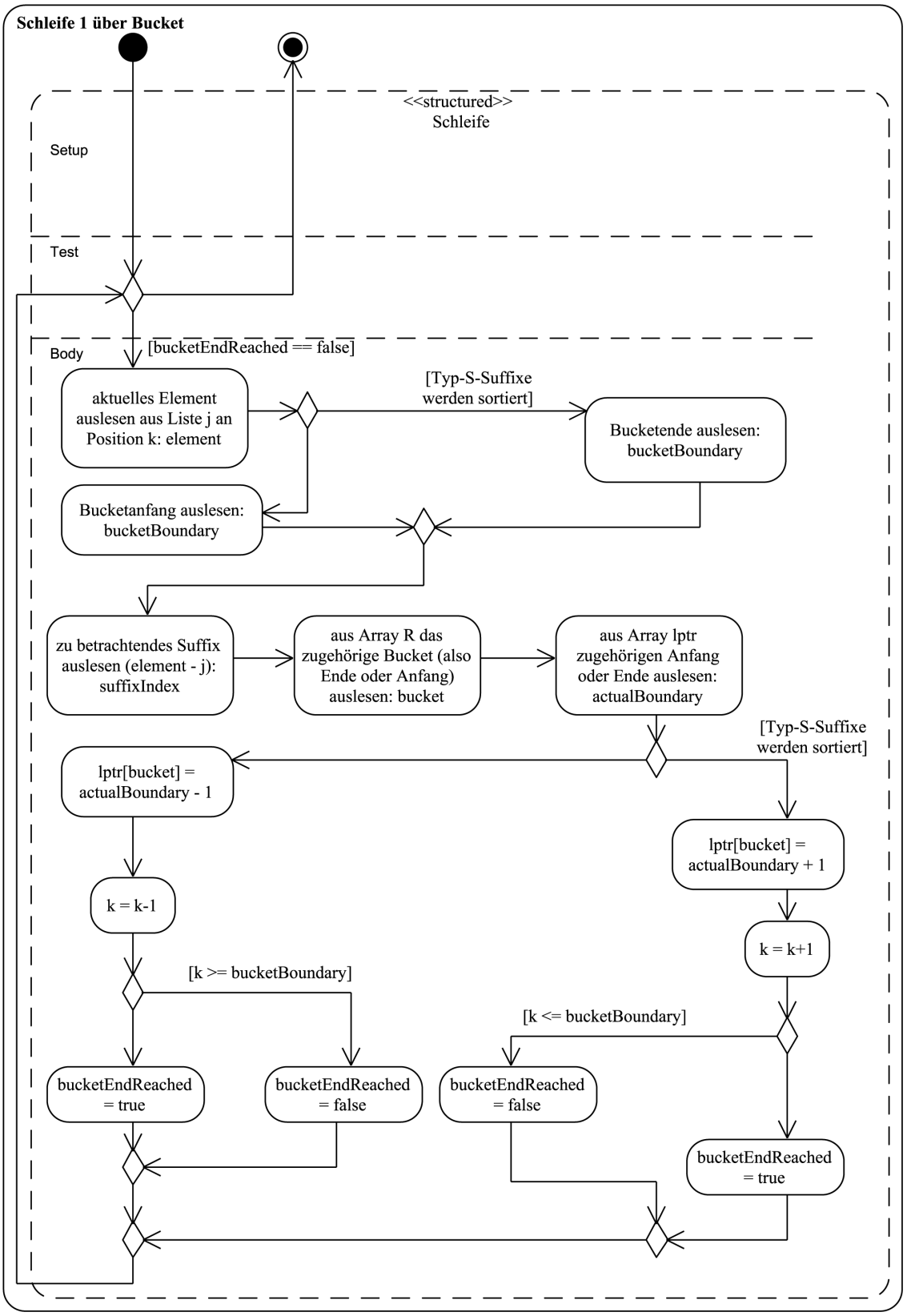


Abbildung 3.17: Erste Schleife über das Bucket

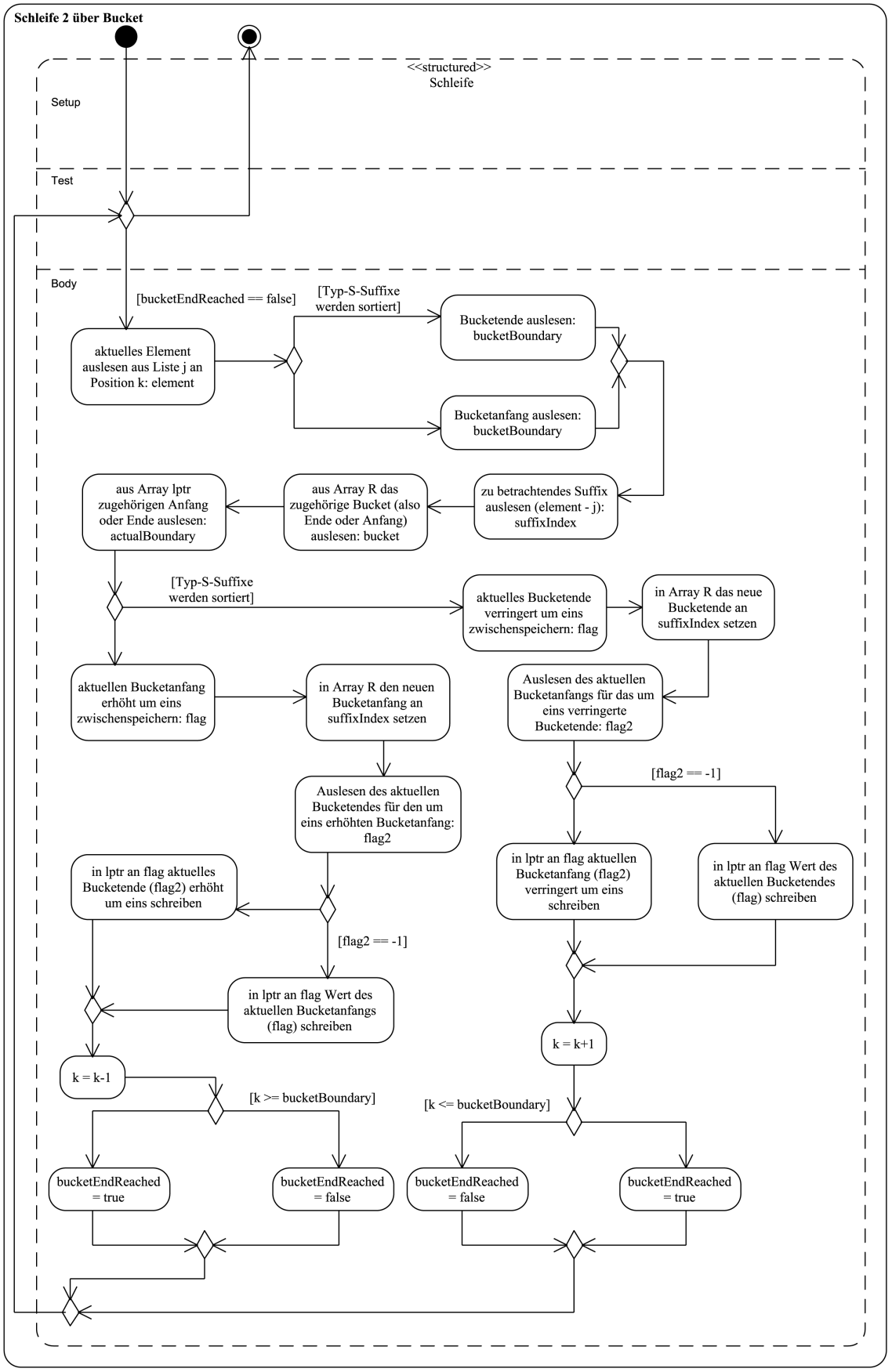


Abbildung 3.18: Zweite Schleife über das Bucket

Sind alle vorhandenen Listen auf diese Weise verarbeitet worden, wird, wenn das Array C noch nicht direkt sortiert war, die Operation `hasEveryBucketOnlyOneString` aufgerufen, die überprüft, ob sich alle Suffixe in einem einzelnen Bucket befinden und das Array C somit sortiert ist oder ob weitere Schritte zum Sortieren nötig sind.
Die Überprüfung muss nicht über einen Zeichenkettenvergleich durchgeführt werden, sondern erfolgt wieder anhand des Arrays *lptr*, wie in Abbildung 3.19 auf S. 171 zu sehen ist. Dafür wird innerhalb der Operation `hasEveryBucketOnlyOneString` zunächst eine Datenstruktur angelegt, die jedes in *lptr* auftretende Bucketende oder jeden auftretenden Bucketanfang speichern soll. Innerhalb einer Schleife wird über das Array *lptr* iteriert, solange das Array noch nicht betrachtete Elemente aufweist und nicht bereits ermittelt wurde, dass sich noch mindestens zwei Suffixe im gleichen Bucket befinden. Letzteres wird durch den Wert der `boolean` Variablen `hasOnlyOneString` abgebildet. Sind die Schleifenbedingungen nicht verletzt, so wird das nächste Element aus *lptr* ausgelesen und sein Wert in `bucketBoundary` gespeichert. Betrachtet man diesen Wert, so können drei Fälle auftreten:

1. **Der Wert ist -1.**
   Das bedeutet, dass mindestens ein Bucket des Arrays C mindestens zwei Elemente enthält. Gleichzeitig bedeutet das, dass das Array C noch nicht sortiert ist. Also wird `hasOnlyOneString` auf `false` gesetzt, was auch ein Beenden der Schleife nach sich zieht. Das Array *lptr* muss also nicht weiter durchlaufen werden, da bereits ein Bucket vorhanden ist, das mindestens noch zwei Suffixe enthält. Die Operation gibt also `false` zurück.

2. **Der Wert ist noch nicht aufgetreten.**
   Es ist ein Wert ungleich -1 vorhanden, der zuvor noch nicht im Array gespeichert war. Um festzustellen, ob dieser Wert noch an anderer Stelle auftritt, wird er in der erwähnten Datenstruktur gespeichert und das nächste Element des Arrays *lptr* betrachtet.

3. **Der Wert ist bereits aufgetreten.**
   Die Variable `bucketBoundary` hat einen Wert, der zuvor schon aufgetreten ist und somit in der Datenstruktur für diese Werte enthalten ist. Das bedeutet, mindestens ein Bucket enthält mehr als ein Suffix, da die Grenzen des Buckets gleich sind. Dann kann die `boolean` Variable, die das angibt, auf `false` gesetzt werden und die Schleife verlassen werden.

Abschließend wird der ermittelte `boolean` Wert zurückgegeben.

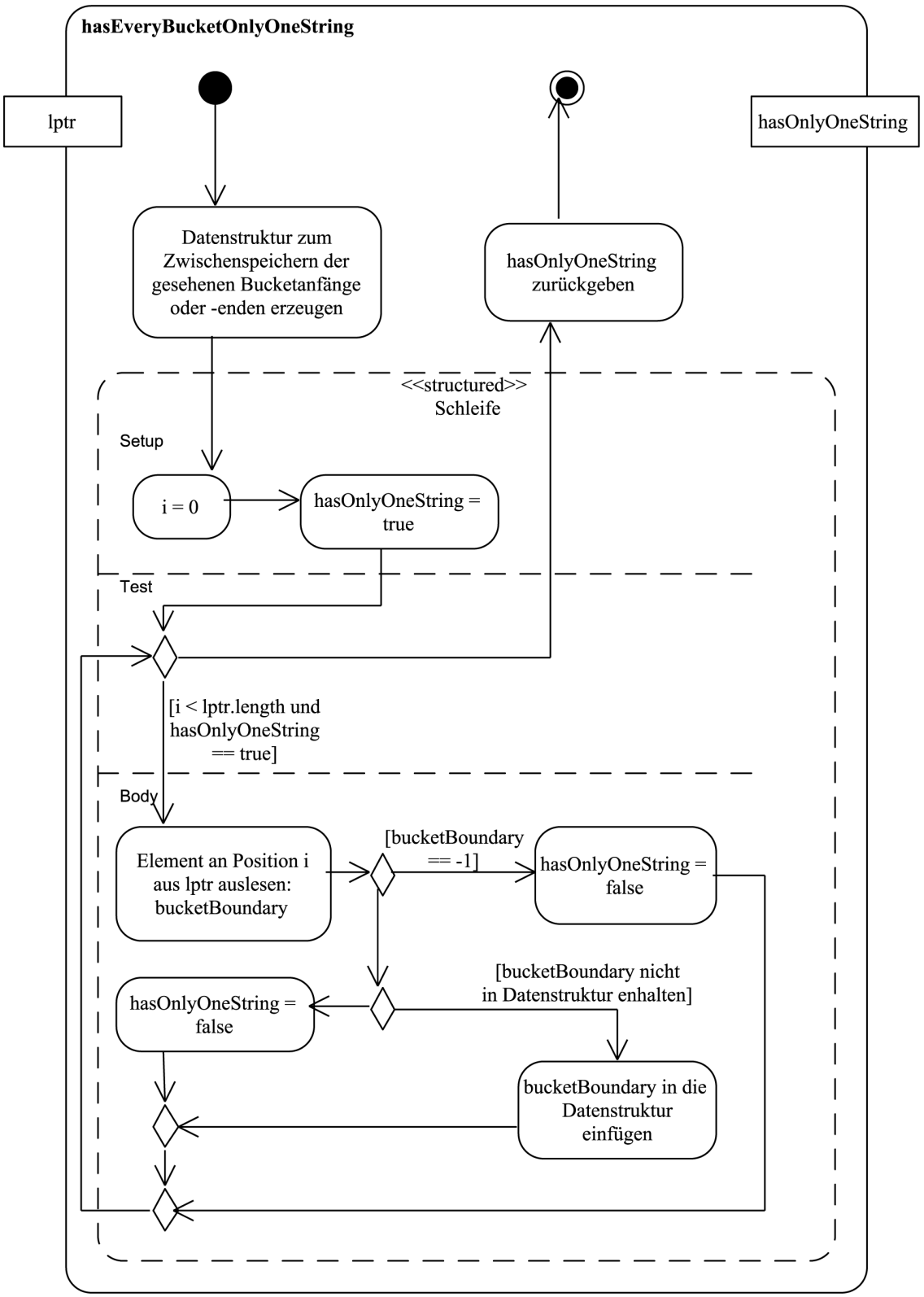


Abbildung 3.19: Ablauf der Operation `hasEveryBucketOnlyOneString`

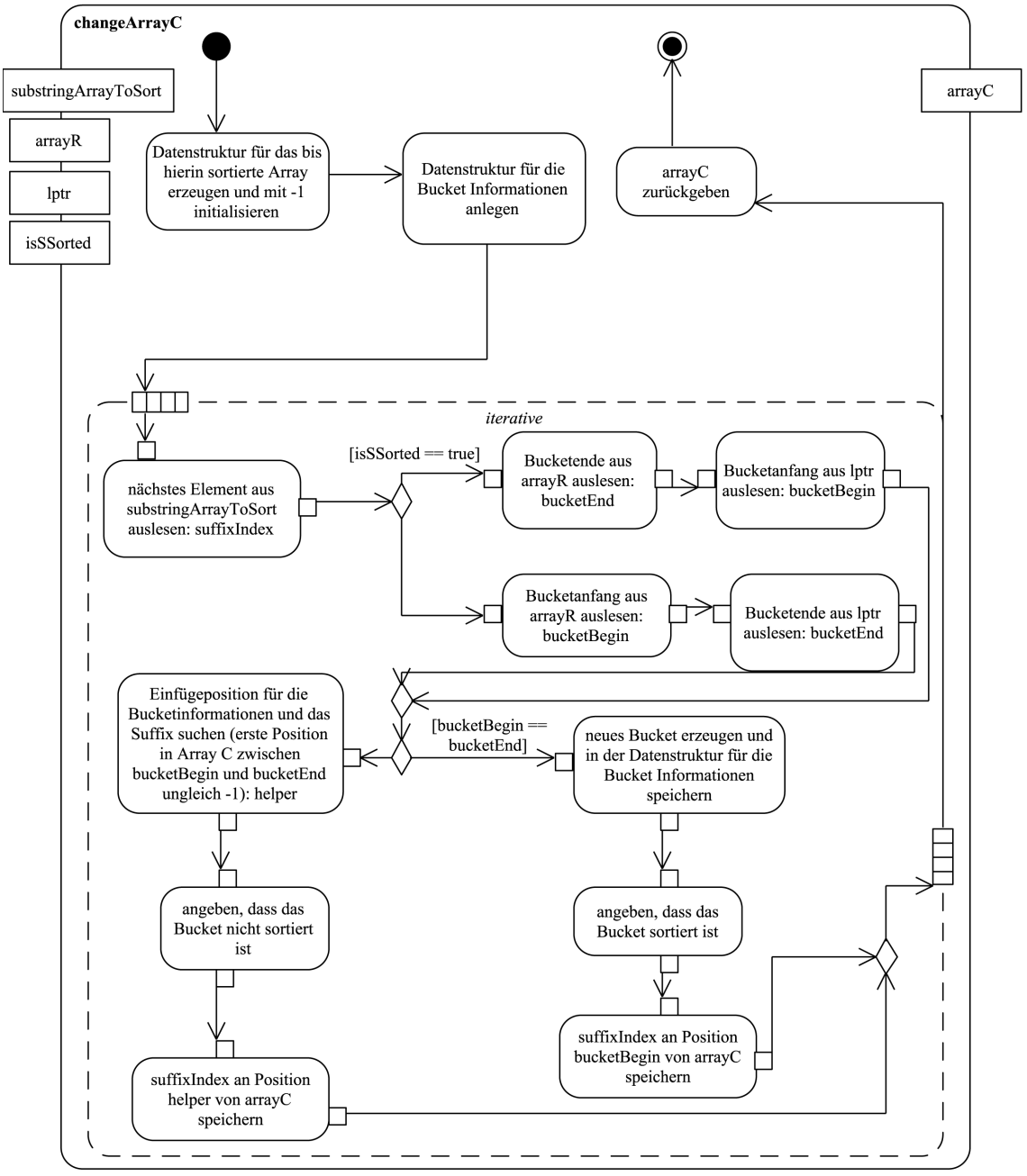


Abbildung 3.20: Ablauf der Operation `changeArrayC`

Die Operation `sortSubstrings` soll das sortierte Array C zurückgeben. Selbst wenn die Operation `hasEveryBucketOnlyOneString` anhand von Array *lptr* ermittelt, dass das Array C sortiert ist, muss zunächst die Sortierung, die durch die Listen erreicht wurde, in Array C abgebildet werden. Auch im anderen Fall, wenn Array C noch nicht sortiert ist, muss die bisherige Sortierung dort abgebildet werden, damit die weiteren Schritte für eine vollständige Sortierung durchgeführt werden können.

Also wird die Operation `changeArrayC` aufgerufen, die die bisherige durch die Listen erreichte Sortierung von Array C mit Hilfe der Arrays R und *lptr* in C abbildet.

Zur Sortierung des Arrays C wird innerhalb der Operation als Erstes eine Datenstruktur für das Array C in seiner bisherigen Sortierung erzeugt und mit -1 initialisiert. Diese Initialisierung ist wichtig, damit Positionen ermittelt werden können, denen noch kein Element zugewiesen wurde. Als Zweites wird eine Datenstruktur für die Bucketinformationen als Klassenvariable erzeugt. Anschließend wird innerhalb einer Schleife über das ursprüngliche Array mit den zu sortierenden Suffixen iteriert. Dabei wird jeweils das nächste Element ausgelesen und das zugehörige Bucketende und der zugehörige Bucketanfang ermittelt. Aus welchem der beiden Arrays - R oder *lptr* - der Anfang und das Ende ausgelesen wird, ist davon abhängig, welcher der beiden Typen sortiert wird. Sind Bucketanfang und Bucketende gleich, so heißt das, dass sich nur ein Suffix im entsprechenden Bucket befindet und dieses Bucket somit sortiert ist. Der Bucketanfang und das Bucketende können also in der Datenstruktur für die Bucketinformationen gespeichert werden. Zusätzlich wird hinterlegt, dass dieses Bucket bereits sortiert ist und bei eventuell nachfolgenden Schritten zur Sortierung von Array C nicht mehr berücksichtigt werden muss. Abschließend wird das Suffix an der Position des Bucketanfangs im neuen Array C gespeichert.

Entsprechen sich dagegen Bucketanfang und Bucketende nicht, so heißt das, dass sich mindestens zwei Suffixe im gleichen Bucket befinden. In diesem Fall muss zunächst die Einfügeposition für das Bucket in der Datenstruktur für die Bucketinformationen und gleichzeitig für das Suffix in Array C gesucht werden. Diese Position ist die erste Position zwischen Bucketanfang und Bucketende, wobei diese beiden Positionen eingeschlossen werden, die im neuen Array C ungleich -1 sind. Dieses Verfahren ist nötig, damit die ursprüngliche Reihenfolge der Suffixe, wie sie im alten Array C vorlag, nicht verändert wird, außer die Suffixe sind eben schon sortiert, dann ist die Einfügeposition aber durch die Bucketgrenzen, die sich entsprechen, eindeutig vorgegeben. Zusätzlich erfolgt die Angabe, dass das entsprechende Bucket noch nicht sortiert ist, und das Suffix wird an der ermittelten Position im neuen Array C gespeichert.

Sind alle Elemente von `substringArrayToSort` auf diese Weise verarbeitet worden, so wird das neue Array C zurückgegeben. Zu sehen ist der Ablauf der Operation in Abbildung 3.20 auf S. 172.

Nun können zwei Fälle eintreten: Zum einen kann festgestellt worden sein, dass die Suffixe im neuen Array C bereits sortiert sind, also jedes Bucket nur ein Suffix enthält, dann wird lediglich das neue Array C zurückgegeben. Zum anderen können die Suffixe im neuen Array C noch nicht sortiert sein, dann müssen sie noch ab-

schließend sortiert werden. Das erfolgt durch die Operation `sortRestOfSuffixes`, zu sehen in Abbildung 3.21 auf S. 175.

Innerhalb der Operation werden zunächst die Bucketinformationen ausgelesen. Das ist wichtig, da die Buckets nun durchnummeriert werden sollen, um laut Algorithmus eine neue Zeichenkette zu erzeugen, die dann auf die gleiche Art sortiert werden soll wie zuvor die eigentliche Zeichenkette. Sind die Bucketinformationen ausgelesen, wird die Operation `setBucketnumbers`, zu sehen in Abbildung 3.22 auf S. 176, aufgerufen. Diese erzeugt ein Array, um dort die Nummer des jeweiligen Buckets zu speichern. Wichtig zu wissen ist, dass gleiche Buckets mehrfach mit den gleichen Informationen gespeichert wurden, wenn sich in diesem Bucket mehrere Suffixe befinden. Pro Suffix existiert also eine Bucketinformation, die aber, wenn sich mehrere Suffixe im gleichen Bucket befinden, gleich ist. Um Nummern für die Buckets zu vergeben, ist es also wichtig, gleichen Buckets auch gleiche Nummern zu geben. Darum wird zunächst die erste Bucketinformation ausgelesen und in der Variablen `lastBucket` gespeichert. Ihr wird die Nummer 0 zugewiesen. Danach wird über die restlichen Bucketinformationen iteriert und jeweils das nächste Element in `actualBucket` gespeichert. Anschließend vergleicht man `lastBucket` und `actualBucket`. Sind sie nicht gleich, so wird die Nummer um eins erhöht, ansonsten bleibt sie gleich. Der aktuellen Position im Array mit den Bucketnummern wird dann die gerade ermittelte Nummer zugewiesen und `lastBucket` auf `actualBucket` gesetzt, so dass das nächste Element mit dem vorherigen verglichen wird. Sind alle Buckets auf diese Weise verarbeitet worden, wird das Array mit den vergebenen Bucketnummern zurückgegeben.

Um nun, wie im Algorithmus vorgegeben, T' erzeugen zu können, müssen die Bucketnummern in der Reihenfolge der zugehörigen Textpositionen aneinandergereiht werden. Aus diesem Grund wird das Array `substringArrayToSort` kopiert, damit Vertauschungen durchgeführt werden können. Das neue Array erhält den Namen `textpositions`. Die darin enthaltenen Textpositionen müssen aufsteigend sortiert werden, um ihre Reihenfolge im ursprünglichen Text widerzuspiegeln. Das wird durch die Operation `q_sort` durchgeführt. Die Operation implementiert einen Quicksort[1], der jedoch nicht nur das Array `textpositions` sortiert, sondern auch die zu den Suffixen gehörenden Bucketnummern in gleicher Weise vertauscht, so dass die Zugehörigkeit erhalten bleibt. Nachdem die vorzusortierenden Suffixe wieder in der Reihenfolge angeordnet sind, in der sie im Text auftauchen, wird in einem extra Array vermerkt, wo sich welches dieser Suffixe in `textpositions` befindet.

1 Vgl. bspw. Balzert (1999), S. 643-647.

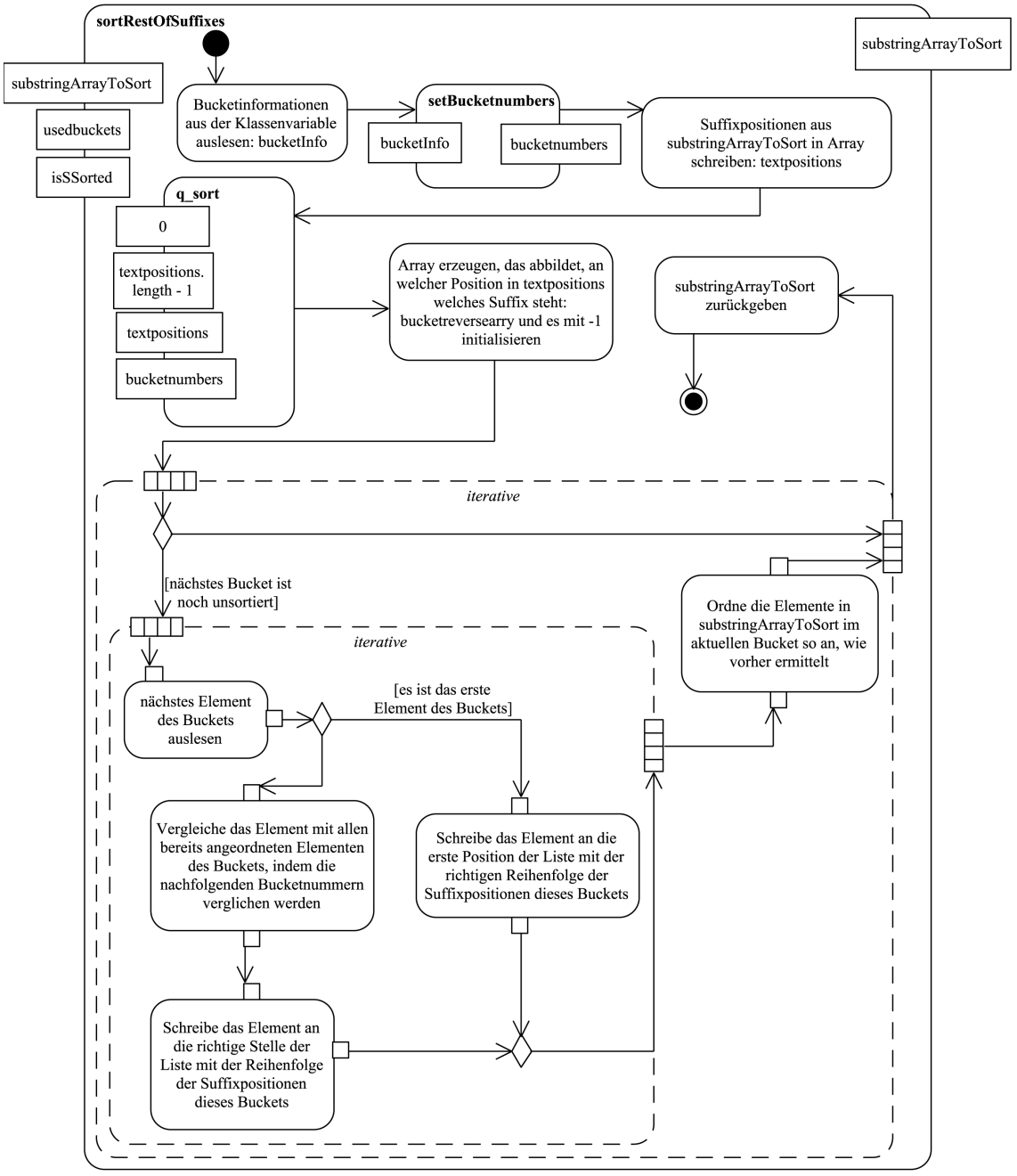


Abbildung 3.21: Ablauf der Operation `sortRestOfSuffixes`

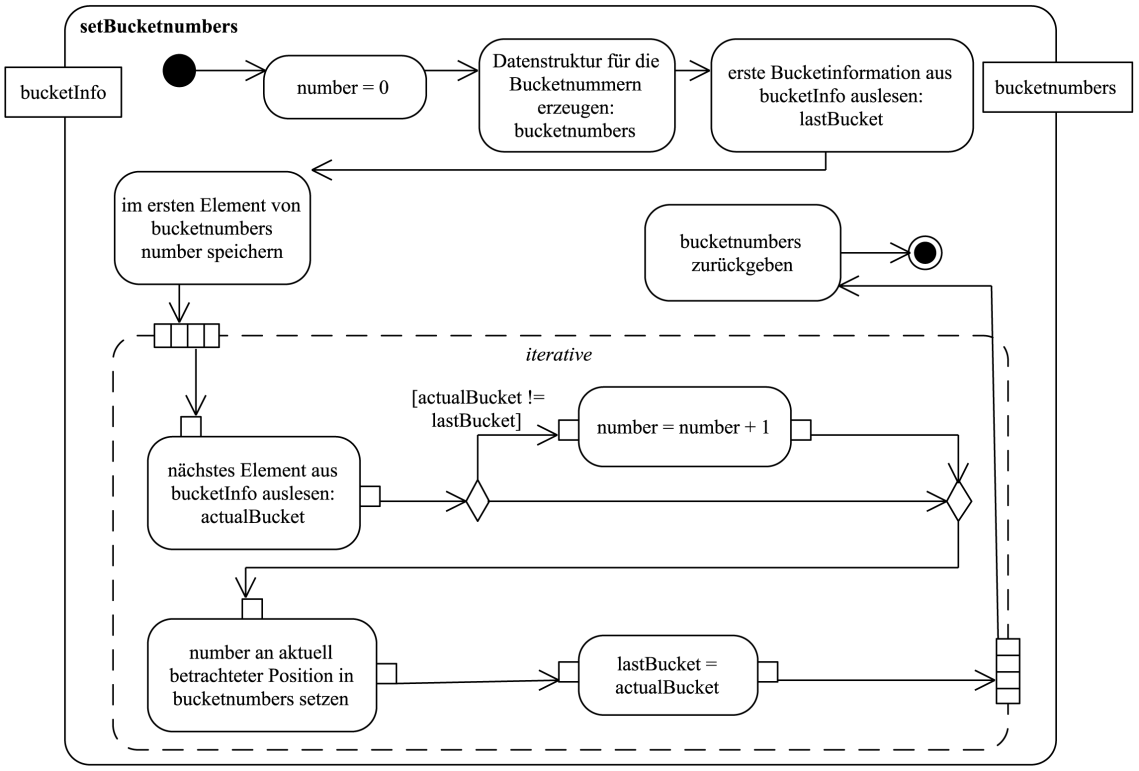


Abbildung 3.22: Ablauf der Operation `setBucketnumbers`

Um die eigentliche Sortierung vorzunehmen, sieht der Algorithmus vor, aus den so sortierten Bucketnummern eine neue Zeichenkette T' zu generieren und diese wie den eigentlichen Text zu sortieren. In der Implementierung der vorliegenden Arbeit wird jedoch ein anderer Ansatz gewählt. Die Zeichenkette T' wird nur implizit gebildet und die Sortierung erfolgt anhand der Betrachtung der jeweils nachfolgenden Bucketnummern. Die Idee dahinter ist, dass man zur Sortierung zweier Teilsuffixe, die sich im gleichen Bucket befinden und somit gleich sind, die nachfolgend beginnenden Teilsuffixe betrachtet. Sind diese bereits sortiert, haben also eine andere Bucketnummer und zudem noch unterschiedliche Bucketnummern, kann man anhand dieser die lexikografische Sortierung der zu sortierenden Suffixe ermitteln. Entscheidend ist hier, dass nur die Buckets sortiert werden müssen, die noch unsortiert sind. Daher wird in der ersten Schleife geprüft, ob das gerade betrachtete Bucket noch unsortiert ist. Ist das der Fall, werden alle Elemente des Buckets betrachtet.

Das erste Element muss noch nicht mit einem anderen Element verglichen werden. Es kann direkt in eine Liste geschrieben werden, die, nachdem das ganze Bucket durchlaufen worden ist, die korrekte Sortierung der im Bucket enthaltenen Suffixe wiedergibt. Beim nächsten Element des Buckets muss dieses dann mit dem ersten Element

verglichen werden. Um keinen Zeichenkettenvergleich durchführen zu müssen, bedient man sich der Idee, die weiter oben erläutert wurde. Man betrachtet sowohl für das aktuelle Suffix als auch für das bereits in der Liste enthaltene Suffix jeweils die Textposition, die als nächstes in der Zeichenkette vorhanden ist und vorsortiert werden soll. Dazu betrachtet man also jeweils das Element aus `textpositions`, das an der Position des aktuellen Suffix erhöht um eins und an der Position des Vergleichssuffixes erhöht um eins steht. Das macht man, da es die Positionen im Text sind, an denen das nächste Teilsuffix beginnt. Diese Teilsuffixe sind die Fortführung der gerade zu vergleichenden Suffixe im Text. Unterscheiden sie sich hinsichtlich ihrer lexikografischen Sortierung, so unterscheiden sich automatisch auch die gerade betrachteten Suffixe. Für diese rechts stehenden Suffixe werden die Bucketnummern ermittelt und miteinander verglichen.
Die Bucketnummern wurden aufsteigend vergeben, das heißt nun Folgendes:

1. Haben die Nachfolgesuffixe unterschiedliche Bucketnummern, dann sind sie bereits lexikografisch sortiert und daran kann abgelesen werden, wie die beiden betrachteten Suffixe sortiert werden müssen:

   a) Ist die Bucketnummer des ersten Vergleichssuffixes kleiner als die des zweiten, so muss das gerade betrachtete Suffix lexikografisch größer sein als das zum Vergleich aus dem gleichen Bucket stammende Suffix. Zur Verdeutlichung nochmals folgende Erläuterungen:

      - Teilsuffixe im gleichen Bucket sind identisch. Es kommt also darauf an, wie der Text nach den Teilsuffixen weitergeht.
      - Kommt nach dem ersten Teilsuffix etwas lexikografisch kleineres als nach dem zweiten, so muss folglich das erste Suffix - und damit das erste Teilsuffix - lexikografisch kleiner sein als das zweite.

   b) Trifft das nicht zu, so muss - mit der gleichen Argumentation - das gerade betrachtete Teilsuffix lexikografisch kleiner sein als das Vergleichssuffix.

2. Haben die nachfolgenden Teilsuffixe ebenfalls gleiche Bucketnummern, so kann an dieser Stelle nicht entschieden werden, wie die lexikografische Sortierung aussieht. Man geht dann über zu den nächsten Vergleichspositionen, bis die maximale Anzahl der Vergleiche erreicht ist. Die nächsten Vergleichspositionen sind dann die Suffixe, die wiederum rechts von den gerade zum Vergleich herangezogenen Suffixen stehen. Die maximale Anzahl von Vergleichen ergibt sich aus der Anzahl an Suffixen, die nach dem aktuell betrachteten oder dem aus dem gleichen Bucket stammenden Suffix noch in Array C vorhanden sind.

Spätestens wenn das eindeutige Endezeichen erreicht ist, lässt sich die lexikografische Sortierung eindeutig festlegen.

Je nachdem, welche Position sich für das aktuell betrachtete Suffix anhand des Vergleichs der Bucketnummern der Nachfolgesuffixe ergibt, wird es in die Liste für die Suffixe des Buckets geschrieben. Danach wird mit dem nächsten Element des Buckets ebenso verfahren, wobei dieses mit allen bereits betrachteten Suffixen verglichen werden muss, um die richtige Einfügeposition zu finden. Sind alle Suffixe des Buckets auf diese Weise verarbeitet worden, werden sie in `substringArrayToSort` in der ermittelten Reihenfolge gespeichert und es wird mit dem nächsten unsortierten Bucket wie beschrieben verfahren, bis alle Buckets abschließend sortiert sind. Dann kann das sortierte Array C zurückgegeben werden.
Als Beispiel, wie die Sortierung innerhalb der Implementierung durchgeführt wird, wird das Beispiel aus Kapitel 3.1.4.3.2.6 nochmals aufgegriffen. Dort ergab sich für die Zeichenkette T =„abrakadabra$“ folgendes Array C mit den entsprechenden Bucketnummern nach der Sortierung durch die m Listen:

	1	2	3	4	5	6	7
C_T	12	1	8	6	4	9	2
	$	a	a	a	a	b	b
		b	b	d	k	r	r
		r	r	a	a	a	a
		a	a	b	d	$	k
Bucketnr.	1	2		3	4	5	6

In der Implementierung wird jetzt das Array *textpositions* erstellt und aufsteigend nach den Textpositionen sortiert, wobei die Bucketnummern den Textpositionen erhalten bleiben. Es ergibt sich folgendes Array *textpositions*:

	1	2	3	4	5	6	7
textpositions	1	2	4	6	8	9	12
Bucketnr.	2	6	4	3	2	5	1

Wie man an Array C erkennen kann, muss nur das Bucket mit Nummer 2 sortiert werden, da alle anderen Buckets nur noch ein Suffix enthalten und somit sortiert sind. Da es sich nur um zwei Suffixe handelt, die sortiert werden müssen, wird das erste, in diesem Fall das Suffix 1, in einer Liste an erster Stelle gespeichert. Danach muss das Suffix 8 daraufhin überprüft werden, ob es vor oder nach Suffix 1 einsortiert

werden muss. Die beiden Teilsuffixe, wie man auch oben an Array C erkennen kann, sind gleich. Es kommt jetzt also darauf an, wie beide Teilsuffixe *nach* ihrem vierten Zeichen weitergehen.
Dazu bestimmt man anhand von Array *textpositions*, welche Suffixe aus Array C in der ursprünglichen Zeichenkette rechts neben Suffix 1 und Suffix 8 stehen. Für Suffix 1 ist das Suffix 2 und für Suffix 8 ist das Suffix 9. Ein Vergleich der Bucketnummern dieser beiden Suffixe führt zu folgendem Ergebnis:

- Die Bucketnummer von Suffix 2 ist 6.
- Die Bucketnummer von Suffix 9 ist 5.
- Die Bucketnummer von Suffix 2 ist somit *größer* als die Bucketnummer von Suffix 9. Das bedeutet, das Suffix an Textposition 2 ist lexikografisch größer als das Suffix an Textposition 9, da diese beiden Suffixe bereits lexikografisch aufsteigend sortiert sind und die Bucketnummern aufsteigend vergeben wurden.
- Des Weiteren bedeutet dies, dass die beiden gleichen Teilsuffixe aus Bucket 2 - Suffix 1 und Suffix 8 - in der Zeichenkette unterschiedlich weitergehen. Suffix 1 wird mit Suffix 2 fortgeführt, also mit etwas lexikografisch größerem als Suffix 8, das mit Suffix 9 weitergeführt wird.
- Somit ist insgesamt Suffix 8 lexikografisch kleiner als Suffix 1 und wird daher in die Liste mit den richtig sortierten Suffixen *vor* Suffix 1 eingeordnet.

Diese abschließende Sortierung der Suffixe des Buckets wird dann in Array C übertragen, so dass genau das Array C, wie in der Beschreibung des Algorithmus dargestellt, resultiert.
Nachdem das Array mit den vorzusortierenden Suffixen lexikografisch aufsteigend sortiert ist, ist die Operation `sortTheLessSubstringSuffixes` beendet. Sie gibt an die Operation `sortArray` das Array mit den vorsortierten Suffixen zurück. Innerhalb der Operation `sortArray` wird nun die Operation `createBucketsWholeArray` aufgerufen, die in Abbildung 3.23 auf S. 180 zu sehen ist.
Diese Operation erzeugt für das gesamte Suffix Array, also das Array A, so, wie es im Moment ist, also alle Suffixe sortiert nach ihrem ersten Zeichen und nach ihrer Reihenfolge in der Zeichenkette, Buckets nach eben diesem ersten Zeichen der enthaltenen Suffixe. Dazu wird das erste Zeichen des aktuellen Suffixes mit dem ersten Zeichen des im Suffix Array links stehenden Suffixes verglichen. Sind diese Zeichen unterschiedlich, so wird ein neues Bucket für das aktuelle Suffix angelegt

und gespeichert. Im anderen Fall wird das Ende des Buckets für das links stehende Suffix um eins nach rechts verschoben. Das Bucket wird ebenfalls gespeichert. Sind auf diese Weise alle Suffixe verarbeitet worden, so wird die Datenstruktur mit den Bucketinformationen zurückgegeben.

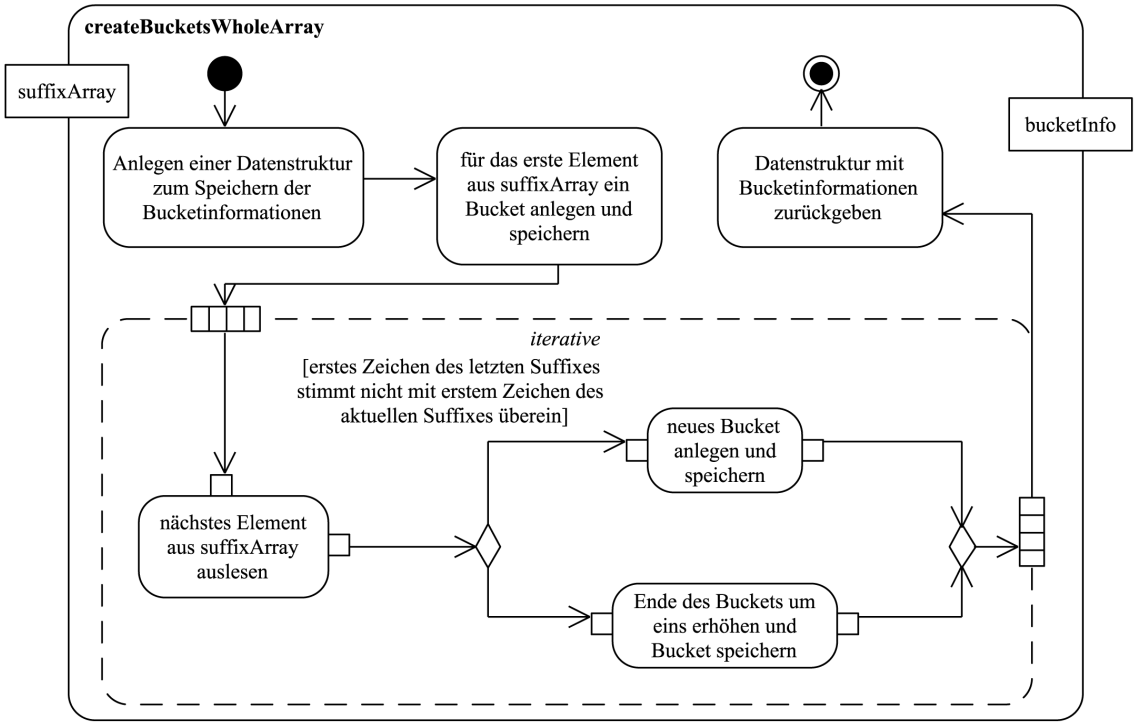


Abbildung 3.23: Ablauf der Operation `createBucketsWholeArray`

Nachdem die kleinere Anzahl an Suffixen vorsortiert wurde und die Buckets für das gesamte Suffix Array in der jetzigen Form aufgrund des ersten Zeichens der enthaltenen Suffixe erzeugt wurden, wird das Suffix Array abschließend sortiert. Das erfolgt je nachdem, welche Suffixe vorsortiert wurden. Wurden Typ-S-Suffixe vorsortiert, so erfolgt der Aufruf der Operation `sortArrayWithS`, zu sehen in Abbildung 3.24 auf S. 181, ansonsten der Aufruf der Operation `sortArrayWithL`.
Der Ablauf der beiden Operationen unterscheidet sich nur dadurch, dass die beteiligten Arrays, einmal das mit den vorsortierten Suffixen und einmal das eigentliche Suffix Array, aus unterschiedlichen Richtungen durchlaufen werden. Im Fall der Typ-S-Suffixe wird das Array mit den vorsortierten Suffixen von rechts nach links durchlaufen und das jeweils aktuelle Suffix an das aktuelle Ende seines Buckets im Suffix Array getauscht, wenn es sich nicht bereits auf der Position des aktuellen Endes befindet. Anschließend wird das aktuelle Ende um eins nach links bewegt. Nach dem Durchlauf über das Array sind alle Typ-S-Suffixe in ihrer lexikografisch rich-

tigen Position im Suffix Array. Gleiches gilt umgekehrt auch für die Typ-L-Suffixe. In diesem Fall wird das Array mit den vorsortierten Suffixen von links nach rechts durchlaufen und das jeweilige Suffix an den aktuellen Anfang seines Buckets im Suffix Array getauscht. Schließlich wird der Anfang um eins nach rechts verschoben.

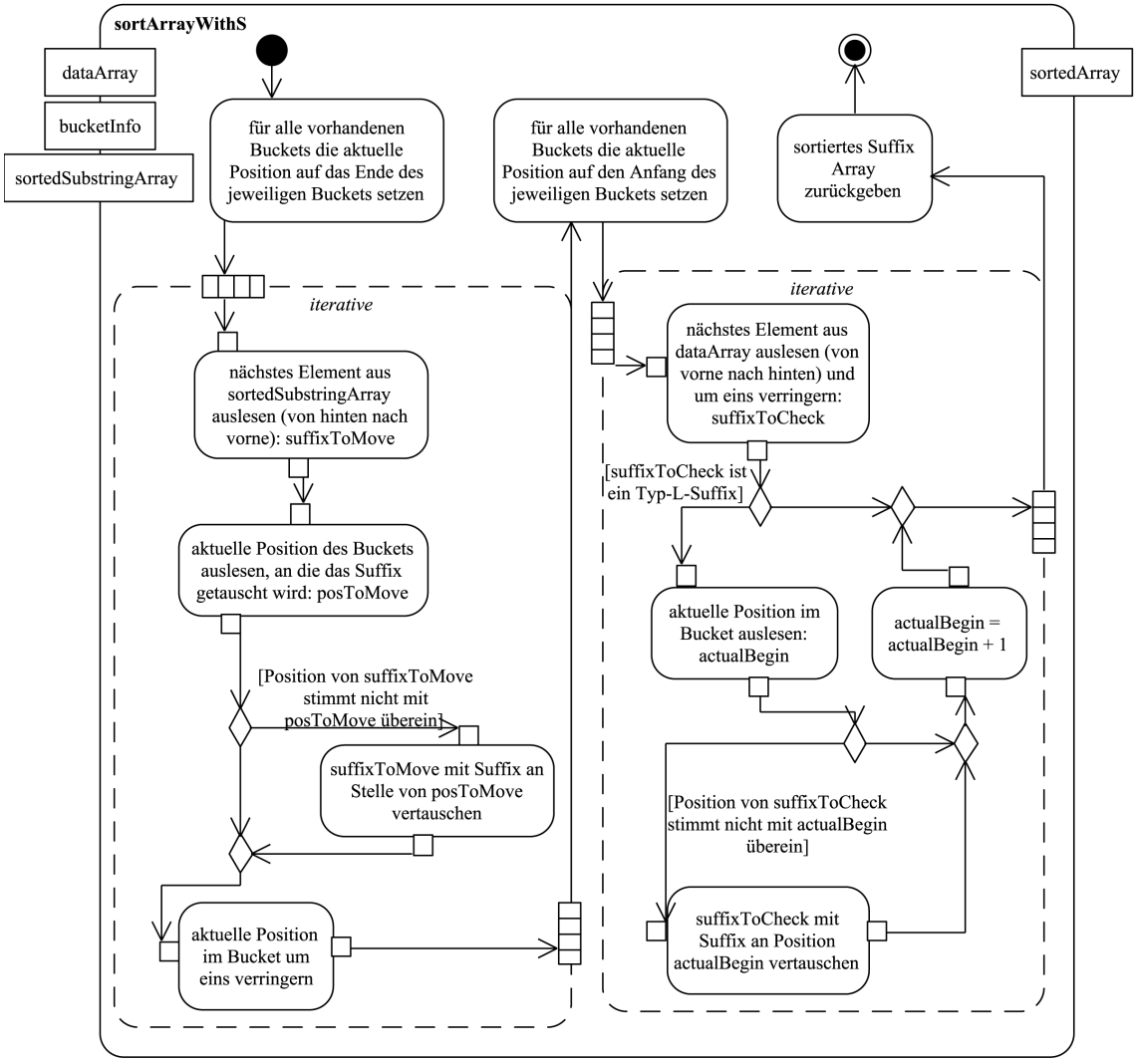


Abbildung 3.24: Ablauf der Operation `sortArrayWithS`

Auch der abschließende Durchlauf über das Suffix Array, bei dem jeweils die Suffixe des Typs sortiert werden, der nicht vorsortiert wurde, unterscheidet sich lediglich durch seine Richtung und dadurch, ob die Suffixe an den aktuellen Anfang oder das aktuelle Ende ihres Bucket getauscht werden und in welche Richtung die aktuelle Position verschoben wird. Ist das gesamte Array auf diese Art durchlaufen, ist es vollständig lexikografisch aufsteigend sortiert und somit fertiggestellt.

3.1.4.5 Anwendungsmöglichkeiten für Suffix Arrays

Die Anwendungsmöglichkeiten für Suffix Arrays entsprechen denen des Suffix Trees. Die bereits genannten Anwendungsmöglichkeiten sind:

- **Suchen nach exakten Übereinstimmungen mit einer vorgegebenen Zeichenkette**
  Verfügt man über ein Suffix Array, das einen bestimmten Text komplett darstellt, und sucht man in diesem Text nach einer Zeichenkette, so kann das nicht, wie bei einem Suffix Tree, direkt festgestellt werden. In diesem Fall muss man bspw. eine binäre Suche[1] anwenden, um das Suffix oder die Suffixe zu finden, die mit der Zeichenkette übereinstimmen.

  Wichtig ist diese Anwendungsmöglichkeit im Kontext der Arbeit, da für die Bildung des Vektors eines Textes, der für das Klassifizieren und das Clustern herangezogen werden kann, die gefundenen Features überprüft werden müssen, ob sie sich im entsprechenden Text befinden. Es wird also eine exakte Übereinstimmung im Text gesucht.[2]

- **Suchen nach teilweisen Übereinstimmungen mit einer vorgegebenen Zeichenkette**
  Ebenso wie in einem Suffix Tree können auch teilweise Übereinstimmungen mit der zu suchenden Zeichenkette gefunden werden. Dafür geht man genauso vor wie im Fall der kompletten Übereinstimmung und prüft dann, inwieweit das gefundene Suffix mit der vorgegebenen Zeichenkette übereinstimmt. Die Position der teilweisen Übereinstimmung ist im Suffix Array gespeichert. Sollte es mehrere Positionen geben, so stehen sie im Suffix Array nebeneinander. Diese Anwendungsmöglichkeit spielt im Rahmen der vorliegenden Arbeit keine Rolle, da die gesuchten Features exakt im Text auftauchen müssen. Jedoch

1 Bei der binären Suche handelt es sich um einen Algorithmus, der auf einem Array arbeitet und den Index des gesuchten Elements im Array zurückliefert oder die Angabe, dass das Element nicht im Array vorhanden ist. Dafür müssen die Elemente des Arrays in ihrer Anordnung einer Ordnungsrelation genügen. Das ist bei einem Suffix Array der Fall. Im Algorithmus selbst wird zunächst das mittlere Element des Arrays betrachtet. Ist es gleich dem gesuchten Element, so wird der entsprechende Index zurückgegeben. Ist es kleiner als das gesuchte Element, so wird das Verfahren auf den Teil des Arrays angewendet, der größere Indizes aufweist als das betrachtete Element. Im umgekehrten Fall wird der Teil des Arrays durchsucht, der kleinere Indizes als das betrachtete Element aufweist. Durch diese Vorgehensweise wird das Array halbiert und nur noch die eine Hälfte betrachtet und diese wiederum halbiert. Das wird so lange wiederholt, bis das gesuchte Element gefunden ist oder klar ist, dass es sich nicht im Array befindet. Eine Darstellung des Algorithmus in Pseudocode befindet sich in Cormen u.a. (2009), S. 799.

2 Eine genaue Erläuterung, wie dies in der vorliegenden Arbeit durchgeführt wird, befindet sich in Kapitel 3.2.

besteht hier die Möglichkeit, in Zukunft auch teilweise Übereinstimmungen der Features mit dem Text in einem Vektor für den Text abzubilden.

- **Suchen nach allen Vorkommen eines bestimmten Musters oder Suchen nach allen Suffixen, die ein bestimmtes Muster enthalten**
  Beide genannten Suchmöglichkeiten sind im Prinzip gleich. Vorgegeben wird ein bestimmtes Muster und man sucht dieses Muster im Suffix Array. Dadurch, dass das Suffix Array die Suffixe des Textes in lexikografisch aufsteigender Reihenfolge enthält, stehen lexikografisch ähnliche Suffixe im Suffix Array nebeneinander. So kann dann, wenn das gesuchte Muster gefunden wurde, gezählt werden, wie häufig es im Array vorhanden ist, indem die nachfolgenden Positionen betrachtet werden. Diese Anwendungsmöglichkeit spielt im Rahmen der vorliegenden Arbeit keine Rolle, da im Vektor für den Text nur gespeichert wird, ob ein Feature im selbigen auftaucht oder nicht. Möchte man jedoch nicht nur das bloße Vorhandensein des Features im Text ermitteln, sondern beim Klassifizieren oder Clustern die Vorkommenshäufigkeit betrachten, so ist die Anpassung daran auch nach der Aufbereitung des Textes als Suffix Array möglich.

- **Suche nach Übereinstimmungen, die Fehler enthalten dürfen**
  Für diesen Fall gilt das Gleiche, das beim Suffix Tree gesagt wurde: Es ist auch mit Suffix Arrays möglich, Übereinstimmungen zu finden, die eine bestimmte Anzahl an Fehlern enthalten dürfen, jedoch spielt das im Kontext der vorliegenden Arbeit keine Rolle.

- **Suchen nach der längsten gemeinsamen Teilzeichenkette zweier oder mehrerer Texte**
  Das Finden der längsten gemeinsamen Teilzeichenkette oder überhaupt gemeinsamer Teilzeichenketten zweier oder mehrerer Texte setzt, wie auch im Fall des Suffix Trees, zunächst ein so genanntes generalisiertes Suffix Array für diese Texte voraus. Ist dieses erstellt, so stehen lexikografisch ähnliche Suffixe der verschiedenen Texte und auch eines Textes im generalisierten Suffix Array nebeneinander. Das bedeutet, sie bilden einen Bereich im generalisierten Suffix Array, der ermittelt werden kann. Dadurch lassen sich gemeinsame Teilzeichenketten finden, die mehrfach in einem Text oder in mehreren Texten vorkommen. Im Kontext der vorliegenden Arbeit bildet diese Anwendungsmöglichkeit ein Herzstück. Mit ihr ist es möglich, Features für die zu klassifizierenden oder zu clusternden Texte zu finden.[1]

1 Siehe Kapitel 3.2.

3.1.5 Verbindung mehrerer einzelner Suffix Arrays zu einem generalisierten Suffix Array

3.1.5.1 Definition generalisiertes Suffix Array

Im Kapitel 3.1.3.4 über Suffix Trees wurde bereits erläutert, was ein generalisierter Suffix Tree ist und wie er erzeugt wird. Auch im Hinblick auf die Verwendung der Datenstruktur Suffix Array muss es eine generalisierte Form dieser Datenstruktur geben, um *mehrere* Texte vorverarbeiten zu können, damit die Featureermittlung mit Hilfe dieser Datenstruktur durchgeführt werden kann.
Ein generalisiertes Suffix Array ist ein Suffix Array, für das die gleichen Voraussetzungen gelten, wie für das Suffix Array eines einzelnen Textes, siehe Definition 10 auf S. 104. Es gilt also auch hier, dass die Suffixe der Texte lexikografisch sortiert in diesem generalisierten Suffix Array enthalten sind und lediglich die Positionen der Suffixe, nicht aber die Suffixe an sich gespeichert werden. Der Unterschied zwischen einem generalisierten Suffix Array und einem Suffix Array für einen einzelnen Text ist der, dass es nicht ausreichend ist, nur die Positionen der Suffixe zu speichern, ohne die Beziehung dieser zum jeweiligen Text aufrecht zu erhalten. Daraus folgt, dass zusätzlich auch eine Identifikation für den Text gespeichert werden muss.
Des Weiteren muss es die Möglichkeit geben, nicht nur eine Kombination aus Text-Identifikation und Suffixposition in einer Position des Suffix Arrays zu speichern, sondern mehrere dieser Kombinationen. Das war vorher bei der Aufbereitung nur eines Textes mit Hilfe eines Suffix Arrays nicht nötig, da ein Text niemals zwei vollkommen identische Suffixe enthalten kann. Werden jedoch mehrere Texte in einem Suffix Array gespeichert, so kann dieser Fall eintreten. Aus diesem Grund muss hier die Datenstruktur erweitert werden, um das zu ermöglichen.

Definition 18. *Ein **generalisiertes** Suffix Array mehrerer Zeichenketten ist eine Liste aller Anfangspositionen der Suffixe der Zeichenketten, wobei sich die Reihenfolge aus der lexikografisch aufsteigenden Sortierung der dazugehörigen Suffixe der Zeichenketten ergibt.*

Definition 19. *Die Anfangspositionen in einem generalisierten Suffix Array bestehen aus zwei Elementen. Das erste Element definiert die Zeichenkette, aus der das Suffix stammt, und das zweite Element definiert die Position des Suffixes in der entsprechenden Zeichenkette. Letztere Angabe entspricht der Anfangsposition eines Suffixes in einem Suffix Array.*

Definition 20. *Der Inhalt eines Feldelements des generalisierten Suffix Arrays kann aus mehreren Anfangspositionen, so wie in Definition 19 definiert, bestehen, wenn ein Suffix in mehreren Zeichenketten enthalten ist.*

Für die beiden Texte aus dem vorherigen Unterkapitel - T_1 =„sitzplatz“ und T_2 =„stehplatz“ - ergibt sich folgendes generalisiertes Suffix Array[1]:

Ergänze die Texte T_1 und T_2 um das Endezeichen „$“.

T_1 = „sitzplatz$“ und T_2 = „stehplatz$“

Nummeriere alle Zeichen der Texte T_1 und T_2 von 1 bis $|T_i|$ mit $i \in \{1, 2\}$ durch.

	1	2	3	4	5	6	7	8	9	10
T_1	s	i	t	z	p	l	a	t	z	$
T_2	s	t	e	h	p	l	a	t	z	$

Füge die Nummern der Suffixe der Texte T_1 und T_2 in lexikografischer Reihenfolge in ein Array ein. Ergänze sie dabei um die ID des Textes, so dass als Element des Arrays Folgendes entsteht: i, s mit $i \in \{1, 2\}$ und $s \in \{1, ..., 10\}$.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
1,10 2,10	1,7 2,7	 2,3	 2,4	1,2	1,6 2,6	1,5 2,5	1,1	 2,1	 2,2	1,8 2,8	1,3	1,9 2,9	1,4
$	a t z $	e h p l a t z $	h p l a t z $	i t z p l a t z $	l a t z $	p l a t z $	s i t z p l a t z $	s t e h p l a t z $	t e h p l a t z $	t z $	t z p l a t z $	z $	z p l a t z $

1 Für die Darstellung eines generalisierten Suffix Arrays in der vorliegenden Arbeit gilt dasselbe, wie für ein Suffix Array gesagt: Die Abbildung erfolgt zwar in Tabellenform, es handelt sich jedoch nicht um eine Tabelle. Aus diesem Grund werden die Zeilen nicht immer beschriftet. Die erste Zeile beinhaltet immer den Index des generalisierten Suffix Arrays. Sie ist durch eine doppelte Linie vom Inhalt des Arrays getrennt.

Zur Veranschaulichung werden die Suffixe der Texte im Beispiel noch unterhalb der doppelten Linie aufgeführt. In der eigentlichen Implementierung wären sie nicht mehr explizit vorhanden, sondern nur noch ihre Textpositionen. Wie bereits zuvor stellen auch hier die kleiner gedruckten Zahlen in der ersten Zeile die Nummerierung der Positionen des generalisierten Suffix Arrays dar. Auch hier wird das Endezeichen „$“ als kleinstes Zeichen des Alphabets behandelt.[1]

3.1.5.2 Algorithmus zum Aufbau eines generalisierten Suffix Arrays

Eine Möglichkeit, ein generalisiertes Suffix Arrays zu erstellen, würde nach dem gleichen Prinzip erfolgen wie die Erzeugung eines Suffix Arrays aus einem Suffix Tree[2]. Der Unterschied bestünde darin, dass zunächst ein generalisierter Suffix Tree aufgebaut werden würde und dann, entsprechend der beschriebenen Methode, ein generalisiertes Suffix Array durch das Durchlaufen des Baumes erzeugt werden könnte. Diese Methode ist jedoch, wie auch schon bei ihrer Vorstellung für die Erzeugung eines Suffix Arrays für einen Text erläutert, nicht sinnvoll. Die Speicherplatzeinsparungen, die sich durch das Verwenden eines Suffix Arrays *anstatt* eines Suffix Trees ergeben, wären auch im generalisierten Fall hinfällig, wenn man zur Erstellung des generalisierten Suffix Arrays zunächst den Suffix Tree erstellen würde.
In der Literatur werden verschiedene Möglichkeiten vorgeschlagen, direkt - ohne den „Umweg“ über einen generalisierten Suffix Tree - ein generalisiertes Suffix Array für eine Menge von Texten zu erzeugen.

1. **Konkatenation der Texte zu *einem* Text und Erzeugung eines Suffix Arrays für diesen neuen Text**
   Die Konkatenation der Texte bedeutet, dass die einzelnen Texte aus der Menge der Texte $T = \{T_1, T_2, ..., T_n\}$ miteinander zu einem Text verbunden werden, so dass folgender Text $T_{konk} = T_1\$T_2\$...T_n\$$ entsteht.[3] Führt man eine solche Verkettung der Texte durch, dann können in der Behandlung der Endezeichen zwischen den einzelnen Texten drei Fälle unterschieden werden:

   a) **Jeder einzelne Text wird mit dem gleichen Endezeichen versehen und dann werden die Texte miteinander verkettet.**
      Diese Vorgehensweise wird als für die Praxis ausreichend dargestellt.[4] Sie hat den Vorteil, dass kein zusätzlicher Vergleich der Endezeichen beim

1 Vgl. Gusfield (1999), S. 149; Ko u.a. (2003), S. 202.
2 Siehe S. 105.
3 Vgl. Jeon u.a. (2005), S. 269; Simpson u.a. (2010), S. i369; Sirén (2009), S. 64; Fischer u.a. (2005), S. 610; Shi (1996), S. 13.
4 Vgl. Fischer u.a. (2005), S. 610.

Aufbau des generalisierten Suffix Arrays erfolgen muss, was der Fall wäre, wenn eine Ordnung für das gleiche Symbol in unterschiedlichen Texten existieren würde. Zusätzlich ist vorteilhaft, dass keine große Menge an zusätzlichen Zeichen, die in keinem der Texte vorkommen dürfen, bereitgestellt werden muss und dass gleiche Suffixe unterschiedlicher Texte im gleichen Feldelement des generalisierten Suffix Arrays gespeichert werden.

b) **Jeder einzelne Text wird mit einem eindeutigen Endezeichen versehen und dann werden die Texte miteinander verkettet.**
Ein eindeutiges Endezeichen bedeutet, dass jeder Text ein unterschiedliches Zeichen als Endezeichen erhält.[1] Vorteilhaft ist, dass so jeder Text auch in dem verketteten Text von den anderen Texten unterscheidbar ist. Nachteilig ist jedoch, dass eine große Menge an unterschiedlichen Zeichen, die als Endezeichen verwendet werden können, bestimmt werden muss und dass gleiche Suffixe nicht mehr im gleichen Feldelement des generalisierten Suffix Arrays gespeichert werden, sondern jedes Suffix des verketteten Textes eindeutig ist.

c) **Jeder einzelne Text wird mit dem gleichen Endezeichen versehen, jedoch erhalten diese Endezeichen eine Ordnung und dann werden die Texte miteinander verkettet.**
Das bedeutet, in diesem Fall wird zwar jeder Text mit dem gleichen Zeichen als Endezeichen versehen, jedoch erfolgt eine Definition, dass das Zeichen, wenn es aus dem ersten Text stammt, kleiner ist als das gleiche Zeichen, wenn es aus dem zweiten Text stammt usw.[2] Vorteilhaft ist wieder, dass sich die Texte unterscheiden lassen, nachteilig ist jedoch, dass eine Ordnung definiert werden und dann auch beim Vergleichen eingehalten werden muss und dass gleiche Suffixe nicht im gleichen Feldelement des generalisierten Suffix Arrays gespeichert werden.

Sind alle Texte zu einem Text verkettet worden, so können unterschiedliche Algorithmen verwendet werden, um aus diesem Text ein Suffix Array zu erstellen. Man verwendet einen der bereits erwähnten Algorithmen, siehe S. 147, der für die Erstellung eines Suffix Arrays eines Textes gedacht ist.

1 Vgl. Sirén (2009), S. 64; Jeon u.a. (2005), S. 269.
2 Vgl. Simpson u.a. (2010), S. i369; Sirén (2009), S. 64; Fischer u.a. (2005), S. 610.

Durch die Verkettung der Texte zu einem Text wird so „automatisch“ das generalisierte Suffix Array erstellt.[1] Zu beachten ist dabei jedoch, dass die Algorithmen die Suffixe der einzelnen Texte betrachten müssen und nicht die Suffixe des verketteten Textes, um ein korrektes generalisiertes Suffix Array liefern zu können.[2]

2. **Zusammenführen (*engl. mergen*) der Suffix Arrays zu *einem* generalisierten Suffix Array**
   Eine andere Möglichkeit ist, die Texte nicht miteinander zu einem Text zu verketten und dann das generalisierte Suffix Array für diesen verketteten Text zu erstellen, sondern für jeden Text ein einzelnes Suffix Array mit einem der erwähnten Algorithmen zu erzeugen und anschließend diese einzelnen Suffix Arrays zu einem generalisierten Suffix Array für alle Texte zusammenzuführen. Für dieses Zusammenführen existieren mehrere Möglichkeiten, die teilweise darauf beruhen, dass kein „normales“ Suffix Array für jeden Text erzeugt wird, sondern eines mit zusätzlichen Informationen:

   - **Zusammenführen zweier komprimierter (engl. *compressed*) Suffix Arrays**
     Anstelle eines Suffix Arrays, wie es bereits beschrieben wurde, siehe S. 103, wird hier für jeden Text ein komprimiertes Suffix Array[3] erzeugt.[4] Das geschieht mit Hilfe einer Burrows-Wheeler-Transformation[5]. Diese wird auch für das Zusammenführen der komprimierten Suffix Arrays zu einem generalisierten Array verwendet.[6] Nachteilig an dieser Art der Erzeugung des generalisierten Suffix Arrays ist, dass komprimierte Suffix Arrays für die einzelnen Texte vorhanden sein müssen.

1 So verwenden Simpson u.a. (2010) eine Abwandlung des Algorithmus von Nong u.a. (2009), vgl. Simpson u.a. (2010), S. i369, der wiederum eine Abwandlung des Algorithmus von Ko u.a. (2003); Ko u.a. (2005) ist. Bedingt durch das Fehlen anderer Algorithmen zum Zeitpunkt der Entstehung des Papers verwendet Shi (1996) dagegen eine Abwandlung des ursprünglichen Algorithmus von Manber u.a. (1990), vgl. Shi (1996), S. 13. Fischer u.a. (2005) konkretisieren nicht, welchen Algorithmus sie zur Erzeugung des generalisierten Suffix Arrays für den verketteten Text benutzen.

2 Das lässt sich an einem Beispiel verdeutlichen: Gegeben seien die Texte $T_1 = abc\#$ und $T_2 = def\$$. Der verkettete Text T_{konk} ist $T_{konk} = abc\#def\$$. Alle Suffixe, die im generalisierten Suffix Array auftauchen dürfen und müssen, sind die folgenden: $\#$, $\$$, $abc\#$, $bc\#$, $c\#$, $def\$$, $ef\$$ und $f\$$. Nicht auftauchen dürfen jedoch beispielsweise: $bc\#def\$$ oder $\#def\$$. Hier würden sonst Suffixe über die Textgrenzen hinaus gebildet, die aber nicht zum generalisierten Suffix Array gehören. Die Algorithmen, die ursprünglich für das Erstellen eines Suffix Arrays eines einzelnen Textes gedacht waren, müssen also dementsprechend angepasst werden.

3 Vgl. Grossi u.a. (2000), S. 398-401.

4 Vgl. Sirén (2009), S. 64-66.

5 Vgl. Burrows u.a. (1994); Adjeroh u.a. (2008), S. 1-14.

6 Vgl. Sirén (2009), S. 66 f.

Es kann auch eine verbesserte Datenstruktur, die *backward search*[1] unterstützt, verwendet werden. Sie entwickelt das komprimierte Suffix Array noch weiter und dient ebenfalls dem schnelleren Auffinden der Suffixe des einen Textes in der Datenstruktur des anderen Textes. Hier gilt derselbe Nachteil wie bereits zuvor: das eine Suffix Array muss in eine Datenstruktur umgewandelt werden, die backward search unterstützt.

- **Zusammenführen zweier erweiterter (engl. *enhanced*) Suffix Arrays**
  Bei dieser Art des Zusammenführens von Suffix Arrays zu einem generalisierten Suffix Array werden ebenfalls für die einzelnen Texte keine Suffix Arrays, wie zuvor vorgestellt, erzeugt. Stattdessen wird in Jeon u.a. (2005)[2] für das Zusammenführen zweier Suffix Arrays zu einem generalisierten Suffix Array für eines der beiden Suffix Arrays eine oder mehrere zusätzliche Datenstrukturen erzeugt, die dabei helfen, die Suffixe des anderen Suffix Arrays schneller im Suffix Array mit den zusätzlichen Datenstrukturen zu finden. Durch dieses schnellere Auffinden ist es dann möglich, die Positionen für das generalisierte Suffix Array zu finden, an denen jeweils Suffixe aus dem einen oder dem anderen Suffix Array für die einzelnen Texte stehen müssen. Die Autoren verwenden dabei die Datenstruktur eines *enhanced Suffix Arrays*, wie in Abouelhoda u.a. (2004) beschrieben. Hier wird das Suffix Array um so genannte Suffix Links ergänzt. Diese dienen in einem Suffix Tree dazu, das Traversieren des Baumes beim Suchen nach Suffixen zu verkürzen[3], und erfüllen im Suffix Array den gleichen Zweck. Der Vorteil dieses Ansatzes ist eine kürzere Laufzeit im Vergleich zum Ansatz der direkten Erstellung eines generalisierten Suffix Arrays aus einer Verkettung aller Texte.[4] Nachteilig ist das Verwenden von zusätzlichen Datenstrukturen.

In der vorliegenden Arbeit wird keine der vorgestellten Möglichkeiten, ein generalisiertes Suffix Array zu erzeugen, verwendet. Das liegt an den erwähnten Nachteilen. Führt man sich die erste Problemstellung dieser Arbeit vor Augen, so soll überprüft

1 Vgl. Ferragina u.a. (2005), S. 559-565, für *backward search* sowie den Rest des Artikels und Ferragina u.a. (2000) und Ferragina u.a. (2001) für die Datenstruktur, die backward search unterstützt.
2 Vgl. Jeon u.a. (2005), S. 273-275.
3 Die Beschreibung in Gusfield (1999), S. 98-101, bezieht sich auf den verkürzten Aufbau des Suffix Trees. Die Suffix Links können aber auch beim späteren Suchen nach Suffixen verwendet werden.
4 Vgl. Jeon u.a. (2005), S. 276.

werden, ob wortübergreifende Features qualitativ bessere Ergebnisse beim Klassifizieren und beim Clustern natürlichsprachlicher Texte liefern. Dazu ist es unumgänglich, wortübergreifende Features zu bestimmen. Der genaue Ablauf, wie die Bestimmung erfolgt, ist in Kapitel 3.2 nachzulesen. Im Kern geht es darum, übereinstimmende Textfragmente mehrerer natürlichsprachlicher Texte zu finden. Dafür eignet sich ein generalisiertes Suffix Array, wie es zuvor in den meisten der vorgestellten Möglichkeiten zur Bildung eines solchen generalisierten Suffix Arrays beschrieben wurde, nur mit zusätzlichem Aufwand bei der Suche nach den Textfragmenten.

Wird ein generalisiertes Suffix Array so erzeugt, dass jeder der enthaltenen Texte mit einem eigenen Endezeichen abgeschlossen wird, so ist bei der Suche nach genau gleichen Textfragmenten ein zusätzlicher Schritt nötig, da solche Textfragmente in diesem Fall nicht im generalisierten Suffix Array dadurch gekennzeichnet sind, dass sie in einem Element des generalisierten Suffix Arrays enthalten sind, sondern in zwei nebeneinanderstehenden Elementen. In diesen Fällen müssen zusätzliche Zeichenvergleiche vorgenommen werden.

Auch die Verwendung von zusätzlichen Datenstrukturen und komprimierten Suffix Arrays bietet sich nur an, wenn es auch gleichzeitig darum geht, die Laufzeit bei der Erzeugung der generalisierten Suffix Arrays zu verkürzen. Das steht in der vorliegenden Arbeit nicht im Vordergrund, sondern vielmehr die Prüfung der Machbarkeit des Klassifizierens und des Clusterns mit wortübergreifenden Features und die Prüfung, ob sich dadurch qualitativ bessere Ergebnisse ergeben. Die genannten Datenstrukturen können in weiteren Arbeiten zum genannten Thema oder in einer Implementierung für eine Klassifizierungs- und Clustersoftware verwendet werden anstatt des in der vorliegenden Arbeit verwendeten Ansatzes, der im Folgenden beschrieben wird.

Von einem Verketten aller Texte und den damit verbundenen Schwierigkeiten bezüglich des Endezeichens wird in der vorliegenden Arbeit abgesehen. Stattdessen wird der Ansatz des Zusammenführens der einzelnen Suffix Arrays zu einem generalisierten Suffix Array, im Folgenden mit GSA abgekürzt, gewählt. Das bedeutet, für jeden Text, der im GSA enthalten sein soll, muss ein einzelnes Suffix Array vorliegen, oder aber, wie in Algorithmus 6 auf S. 107 beschrieben, erzeugt werden. Das heißt auch gleichzeitig, dass jeder Text mit dem gleichen Endezeichen versehen wird, um zu ermöglichen, dass gleiche Suffixe im gleichen Element des GSA gespeichert werden. Das Zusammenführen selbst, wie es in der vorliegenden Arbeit durchgeführt wird, basiert auf dem einfachen Algorithmus zum Zusammenführen zweier Arrays[1],

1 Dieser Algorithmus ist der so genannte *Merge Sort* und davon wird die *Combine*-Methode verwendet, vgl. Cormen u.a. (2009), S. 30 f.; Abbildung auf S. 35.

erweitert diesen aber, um die benötigte Zeit zur Erstellung des GSA zu verkürzen. Der einfache Algorithmus durchläuft parallel die beiden zusammenzuführenden Arrays und vergleicht jeweils die aktuellen Elemente miteinander. Da in diesem Fall eine lexikografisch aufsteigende Sortierung des GSA erreicht werden soll, werden die beiden Suffixe lexikografisch miteinander verglichen. Das jeweils kleinere Suffix wird dann in das gemeinsame Array an die nächste Position geschrieben und das nächste Element des Arrays, aus welchem das eingefügte Element stammt, betrachtet. Da beide Arrays lexikografisch aufsteigend sortiert sind, ergibt sich beim gemeinsamen Array ebenfalls eine lexikografisch aufsteigende Sortierung der Elemente. Sind beide aktuellen Elemente gleich, so werden sie gemeinsam an die nächste Position des gemeinsamen Arrays geschrieben und in beiden zusammenzuführenden Arrays wird das nächste Element betrachtet.

Der einfache Algorithmus im Pseudocode sieht wie folgt aus:

Algorithmus 20 erstelleGeneralisiertesSuffixArray (einfach)

Eingabe: Suffix Array SA_{T_1} für T_1 und Suffix Array SA_{T_2} für T_2

Ausgabe: Generalisiertes Suffix Array GSA für die Texte T_1 und T_2

1: Erzeuge ein leeres generalisiertes Suffix Array, das ist GSA
2: $i = 1$, $j = 1$, $k = 1$
3: **while** $i \leq |T_1| \wedge j \leq |T_2|$ **do**
4: Lies Element an Position i aus SA_{T_1} aus, das ist x
5: Lies Element an Position j aus SA_{T_2} aus, das ist y
6: **if** $T_1[x] \prec T_2[y]$ **then**
7: Schreibe Kombination aus ID von T_1 und x an Position $GSA[k]$
8: $i = i + 1$
9: **else if** $T_1[x] \succ T_2[y]$ **then**
10: Schreibe Kombination aus ID von T_2 und y an Position $GSA[k]$
11: $j = j + 1$
12: **else**
13: Schreibe Kombination aus ID von T_1 und x an Position $GSA[k]$
14: Ergänze Kombination aus ID von T_2 und y an Position $GSA[k]$
15: $i = i + 1$
16: $j = j + 1$
17: **end if**
18: $k = k + 1$
19: **end while**

Algorithmus 20 erstelleGeneralisiertesSuffixArray (einfach Teil 2)

20: **if** $i \leq |T_1|$ **then**
21: Schreibe alle noch vorhandenen Suffixe aus SA_{T_1} ans Ende von GSA
22: **else if** $j \leq |T_2|$ **then**
23: Schreibe alle noch vorhandenen Suffixe aus SA_{T_2} ans Ende von GSA
24: **end if**

Für die beiden Beispieltexte T_1 =„sitzplatz$“ und T_2 =„stehplatz$“ ergibt sich folgender Ablauf:
Das Suffix Array für T_1 sieht wie folgt aus:

	1	2	3	4	5	6	7	8	9	10
SA_{T_1}	10	7	2	6	5	1	8	3	9	4
	$	a	i	l	p	s	t	t	z	z
		t	t	a	l	i	z	z	$	p
		z	z	t	a	t	$	p		l
		$	p	z	t	z		l		a
			l	$	z	p		a		t
			a		$	l		t		z
			t			a		z		$
			z			t		$		
			$			z				
						$				

Das Suffix Array für T_2 ist das folgende:

	1	2	3	4	5	6	7	8	9	10
SA_{T_2}	10	7	3	4	6	5	1	2	8	9
	$	a	e	h	l	p	s	t	t	z
		t	h	p	a	l	t	e	z	$
		z	p	l	t	a	e	h	$	
		$	l	a	z	t	h	p		
			a	t	$	z	p	l		
			t	z		$	l	a		
			z	$			a	t		
			$				t	z		
							z	$		
							$			

Der Ablauf der Erzeugung des Arrays GSA, der sich aus Algorithmus 21 ergibt, ist der folgende:
$i = 1$
$j = 1$
$k = 1$
$x = SA_{T_1}[i] = SA_{T_1}[1] = 10$

$y = SA_{T_2}[j] = SA_{T_2}[1] = 10$
$T_1[x] = T_1[10] = \$ = T_2[y] = T_2[10] = \$$
$\Rightarrow$ beide Suffixe sind gleich
$GSA[k] = GSA[1] = 1, x = 1, 10$ und $2, y = 2, 10$

Das GSA für die beiden Texte sieht also zu diesem Zeitpunkt wie folgt aus:

1
1,10 2,10
\$

$i = 2$
$j = 2$
$k = 2$
$x = SA_{T_1}[i] = SA_{T_1}[2] = 7$
$y = SA_{T_2}[j] = SA_{T_2}[2] = 7$
$T_1[x] = T_1[10] =$ atz\$ $= T_2[y] = T_2[7] =$ atz\$
$\Rightarrow$ beide Suffixe sind gleich
$GSA[k] = GSA[2] = 1, x = 1, 7$ und $2, y = 2, 7$
$i = 3$
$j = 3$
$k = 3$
$x = SA_{T_1}[i] = SA_{T_1}[3] = 2$
$y = SA_{T_2}[j] = SA_{T_2}[3] = 3$
$T_1[x] = T_1[2] =$ itzplatz\$ $\succ T_2[y] = T_2[3] =$ ehplatz\$
$\Rightarrow$ Suffix aus T_2 ergänzen
$GSA[k] = GSA[3] = 2, y = 2, 3$
...
$i = 9$
$j = 10$
$k = 13$
$x = SA_{T_1}[i] = SA_{T_1}[9] = 9$
$y = SA_{T_2}[j] = SA_{T_2}[10] = 9$
$T_1[x] = T_1[9] =$ z\$ $\succ T_2[y] = T_2[10] =$ z\$
$\Rightarrow$ beide Suffixe sind gleich
$GSA[k] = GSA[13] = 1, x = 1, 9$ und $2, y = 2, 9$

Jetzt ist die Bedingung der while-Schleife nicht mehr erfüllt, da $j > |T_2|$. Daher wird die Schleife beendet und die verbleibenden Positionen aus T_1 werden an das GSA angehängt. Es ergibt sich also genau das GSA, das auf S. 185 zu sehen ist. Würde nun ein dritter Text diesem GSA hinzugefügt, so verläuft der Algorithmus genauso wie beschrieben, nur dass anstatt des zweiten Suffix Arrays das GSA der zwei ersten Texte verwendet würde und das Verschmelzen mit dem einzelnen Suffix Array in wieder einem neuen GSA gespeichert wird. Für jeden weiteren Text, der hinzukommt, ist der Ablauf genauso.
Überlegt man sich nun, dass die Texte im Gegensatz zu den hier verwendeten Beispieltexten um ein Vielfaches länger sind und bei dieser Art der Erzeugung eines GSA immer auch das bereits vorhandene GSA komplett durchlaufen wird und für jede Position ein Zeichenkettenvergleich durchgeführt wird, so kann man sich vorstellen, dass diese Art der Erstellung eines GSA mit einem hohen Zeitaufwand verbunden ist. Um diesen zu verringern, macht man sich in der verbesserten Form des Algorithmus zu Nutze, dass beide Arrays lexikografisch sortiert vorliegen und es weniger aufwändig ist, das kürzere Suffix Array, also das Suffix Array für den einzelnen Text, komplett zu durchlaufen und das GSA nicht. Im GSA muss nicht jedes Suffix mit den neu einzufügenden Suffixen verglichen werden, sondern immer nur ein kleiner Bereich, nämlich der, der mit dem gleichen Zeichen beginnt.
Um diese zusätzlichen Informationen zu speichern und sie verwenden zu können, im GSA zu „springen“ und so den Suchbereich für die Einfügeposition zu verkleinern, werden zwei Indizes angelegt.[1] Der eine verwaltet die Positionen, an denen die Anfangszeichen der Suffixe im GSA beginnen. Dabei werden nur Zeichen gespeichert, die auch in den Texten des GSA auftauchen. Des Weiteren wird ein Index für den Anfang und das Ende der ersten drei Zeichen, die in den Suffixen der Texte des GSA auftreten, gespeichert. So kann die Suche nach dem Einfügebereich weiter eingegrenzt werden.
Als Beispiel für die Suche mit einem Drei-Zeichen-Index soll das Suffix „info$“ in das GSA mit den Zeichenketten „informatik$“ und „information$“ eingefügt werden.

1 In der Literatur wird ein solcher Index als *supraindex* oder *supra-index* bezeichnet, vgl. bspw. Navarro u.a. (2000), S. 218, und Baeza-Yates u.a. (1999), S. 201 f.

Das GSA sieht folgendermaßen aus:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1,11 2,12	1,7	2,7	1,3	2,3	1,9	1,1	2,1	2,9	1,10	1,6	2,6	1,2	2,2	2,11	2,10	1,4	2,4	1,5	2,5	1,8	2,8
$	a t i k $	a t i o n $	f o r m a t i k $	f o r m a t i o n $	i k $	i n f o r m a t i k $	i n f o r m a t i o n $	i o n $	k $	m a t i k $	m a t i o n $	n f o r m a t i k $	n f o r m a t i o n $	n $	o n $	o r m a t i k $	o r m a t i o n $	r m a t i k $	r m a t i o n $	t i k $	t i o n $

Der Ein-Zeichen-Index ist:

Zeichen	Position
$	1
a	2
f	4
i	6
k	10
m	11
n	13
o	16
r	19
t	21

Der Drei-Zeichen-Index ist:

Zeichen	Anfangsposition	Endposition
ati	2	3
for	4	5
ik$	6	6
inf	7	8
ion	9	9
mat	11	12
nfo	13	14
on$	16	16
orm	17	18
rma	19	20
tik	21	21
tio	22	22

Ermittelt man nun den Einfügebereich für „info$“, so liest man zunächst aus dem Ein-Zeichen-Index die Startposition der Suffixe, die im GSA vorhanden sind und mit „i“ beginnen, aus. Das ist die Position 6. Nun wird überprüft, ob die ersten drei Zeichen des Suffix „info$“ ebenfalls im GSA vorhanden sind, indem sie im Drei-Zeichen-Index „nachgeschlagen“ werden. So wird ermittelt, dass alle Suffixe im GSA, die mit den drei Zeichen „inf“ beginnen, ab Position 7 bis Position 8 im GSA zu finden sind. Für das Einfügen bedeutet das, dass der Einfügebereich auf zwei Positionen beschränkt wird, im Gegensatz zu vorher vier Positionen, wenn nur der Ein-Zeichen-Index vorhanden wäre.

Das führt zu folgendem Ablauf: Soll ein Suffix des einzelnen Suffix Arrays in das GSA eingefügt werden, so wird zunächst geprüft, ob das erste Zeichen des einzufügenden Suffixes bereits im GSA vorhanden ist.

1. **Das erste Zeichen ist vorhanden.**
   Wenn das erste Zeichen bereits vorhanden ist, kann der Suchbereich weiter eingegrenzt werden, wenn auch die ersten drei Zeichen bereits im GSA vorhanden sind. Also wird überprüft, ob die ersten drei Zeichen des einzufügenden Suffixes vorhanden sind.

   a) **Die ersten drei Zeichen sind vorhanden.**
      In diesem Fall muss nur der Bereich des GSA, der zwischen dem Anfangs- und Endindex für diesen Bereich liegt, durchsucht werden. Das bedeutet, es findet ein Zeichenkettenvergleich der Suffixe des GSA vom Anfangsindex mit dem einzufügenden Suffix bis zur korrekten Einfügeposition statt.

   b) **Die ersten drei Zeichen sind nicht vorhanden.**
      Tritt dieser Fall ein, muss der Bereich zwischen dem Anfangsindex für das erste Zeichen und dem Anfangsindex für das nächste Zeichen im GSA durchsucht werden.

2. **Das erste Zeichen ist nicht vorhanden.**
   Ist das erste Zeichen des Suffixes nicht vorhanden, so können auch die ersten drei Zeichen nicht vorhanden sein. Der Suchbereich kann also nicht weiter eingeschränkt werden. Trotzdem muss nicht von Anfang des GSA oder ausgehend von der letzten Einfügeposition des vorangegangenen Suffixes des einzelnen Suffix Arrays gesucht werden. Stattdessen macht man sich die lexikografische Sortierung zu Nutze und sucht das nächstgrößere im GSA vorhandene Zeichen. Das einzufügende Suffix muss dann *davor* eingefügt werden.[1]

Als verbesserter Algorithmus ergibt sich der folgende:

Algorithmus 21 erstelleGeneralisiertesSuffixArray

Eingabe: Suffix Array SA_{T_1} für T_1 und Suffix Array SA_{T_2} für T_2

Ausgabe: Generalisiertes Suffix Array für die Texte T_1 und T_2

1: Betrachte das Suffix Array SA_{T_1} als generalisiertes Suffix Array GSA und erzeuge die drei Indexstrukturen

1 Bei der Implementierungsbeschreibung, siehe Kapitel 3.1.5.3, wird das Verfahren etwas genauer beschrieben.

Algorithmus 21 erstelleGeneralisiertesSuffixArray (Teil 2)

2: $i = 1$
3: $posGSA = 1$
4: **while** $posGSA \leq GSA.length \wedge i \leq |T_2|$ **do**
5: Lies Element an Position i aus SA_{T_2} aus, das ist x
6: Suche die Position in GSA auf die der Ein-Zeichen-Index zeigt, das ist y
7: **if** $y == -1$ **then** ▷ erstes Zeichen ist nicht vorhanden
8: Lies aus dem Index die Position des nächstgrößeren Zeichens in GSA aus, das ist z
9: Füge x vor Position z im GSA ein
10: $posGSA = z$
11: **else**
12: Suche die Position in GSA, auf die der Drei-Zeichen-Index zeigt, das ist z
13: **if** $z == -1$ **then** ▷ ersten drei Zeichen sind nicht vorhanden
14: Suche im Bereich zwischen y und dem Anfang des Bereichs für das nächstgrößere Zeichen die Einfügeposition und füge das Suffix ein, das ist Position q
15: $posGSA = q$
16: **else**
17: Lies das Ende des Drei-Zeichen-Bereichs aus, das ist z'
18: Suche im Bereich zwischen z und z' die Einfügeposition und füge das Suffix ein, das ist q
19: $posGSA = q$
20: **end if**
21: **end if**
22: $i = i + 1$
23: **end while**

Nimmt man an, dass T_1 das GSA ist, so ergibt sich folgender Ablauf für die verbesserte Algorithmusversion:
$i = 1$
$posGSA = 1$
$x = SA_{T_2}[i] = SA_{T_2}[1] = 10$
Im Ein-Zeichen-Index „$“ suchen $y = 1$
Im Drei-Zeichen-Index „$“ suchen $z = 1$ und $z' = 1$
Einfügeposition liegt zwischen 1 und 1, also Suffixvergleich zwischen $GSA[1]$ und $T_2[10]$

$\Rightarrow$ beide Suffixe sind gleich
Füge Kombination aus ID von $2, 10$ $GSA[1]$ hinzu
$posGSA = q = 1$
Das GSA für die beiden Texte zu diesem Zeitpunkt sieht also wie folgt aus:

1
1,10
2,10
$

$i = 2$
$x = SA_{T_2}[i] = SA_{T_2}[2] = 7$
Im Ein-Zeichen-Index „a“ suchen $y = 2$
Im Drei-Zeichen-Index „atz“ suchen $z = 2$ und $z' = 2$
Einfügeposition liegt zwischen 2 und 2,
also Suffixvergleich zwischen $GSA[2]$ und $T_2[7]$
$\Rightarrow$ beide Suffixe sind gleich
Füge $2, 7$ $GSA[2]$ hinzu
$posGSA = q = 2$
$i = 3$
$x = SA_{T_2}[i] = SA_{T_2}[3] = 3$
Im Ein-Zeichen-Index „e“ suchen $y = -1$
nächstgrößeres Zeichen im GSA ist „i“ $z = 3$
Einfügeposition liegt vor 3, es ist kein Suffixvergleich nötig
Füge $2, 3$ zwischen $GSA[2]$ und $GSA[3]$ ein
$posGSA = 3$
...
Nachdem alle Elemente des Suffix Arrays SA_{T_2} in das GSA eingefügt wurden, ergibt sich das bereits bekannte GSA.

3.1.5.3 Implementierung eines generalisierten Suffix Arrays

3.1.5.3.1 Datenstruktur des generalisierten Suffix Arrays

Ein generalisiertes Suffix Array ist in der Implementierung in der vorliegenden Arbeit eine doppelt verkettete Liste. Das bedeutet, jedes Element des GSA speichert seine eigenen Daten, einen Zeiger auf seinen Vorgänger und einen Zeiger auf seinen Nachfolger. Die Elemente werden als Knoten bezeichnet. Die Daten eines Knotens sind alle Kombinationen aus Text-ID und Position des Suffixes in diesem Text, die

sich auf das Suffix, welches an diese Stelle im GSA gehört, beziehen.[1] Die Verwendung einer doppelt verketteten Liste bietet den Vorteil, dass an beliebiger Stelle in der Liste Knoten eingefügt werden können und lediglich die betroffenen Vorgänger- und Nachfolgerzeiger angepasst werden müssen. Beim Verschmelzen eines GSA mit einem Suffix Array für einen einzelnen Text wird genau diese Eigenschaft benötigt.

3.1.5.3.2 Erzeugung eines generalisierten Suffix Arrays

Um ein generalisiertes Suffix Array für zwei oder mehr Texte zu erstellen, für die bereits einzeln jeweils ein Suffix Array existiert[2], werden zwei Fälle unterschieden:

1. Das generalisierte Suffix Array ist leer.
2. Im generalisierten Suffix Array befindet sich mindestens ein Text.

Innerhalb der Implementierung bedeutet das, dass im ersten Fall die Datenstruktur des Suffix Arrays für den einzelnen Text in die Datenstruktur für das generalisierte Suffix Array umgewandelt wird und dass im zweiten Fall das jeweils nächste, zum generalisierten Suffix Array hinzuzufügende Suffix Array mit dem bereits existierenden generalisierten Suffix Array verschmolzen wird.
Um diesen Prozess insgesamt zu starten, wird die Operation `addArrayWithDoubleIndex` aufgerufen. Sie erhält die Parameter `array`, `overallFileID` und `executionID`. Innerhalb der Operation wird zunächst geprüft, ob im generalisierten Suffix Array bereits Texte vorhanden sind. Ist das nicht der Fall, so erfolgt der Aufruf der Operation `convertSuffixArrayIntoMultipleWithDoubleIndex`, ansonsten der Aufruf der Operation `mergeArraysWithDoubleIndex`. Abschließend gibt die Operation die Anzahl der bereits im generalisierten Suffix Array enthaltenen Texte zurück. Ein Diagramm des Ablaufs befindet sich in Abbildung 3.25 auf S. 200.
Wird die Operation `convertSuffixArrayIntoMultipleWithDoubleIndex` aufgerufen, ist also noch kein Text im generalisierten Suffix Array vorhanden, so erfolgt zunächst die Erzeugung der benötigten Zeigerstrukturen für den Index. Jedes generalisierte Suffix Array verfügt über drei Indizes:

1. **einen Index für das erste Zeichen der im generalisierten Suffix Array enthaltenen Suffixe**
   Dieser Index besteht aus zwei Teilen: (1) dem ersten Zeichen des Suffixes, das momentan das lexikografisch kleinste Suffix mit diesem Anfangszeichen ist,

1 Zur Erinnerung: In einem GSA, wie es in der vorliegenden Arbeit definiert ist, sind gleiche Suffixe im gleichen Element des GSA gespeichert.
2 Ansonsten müsste dieses zunächst erstellt werden.

und (2) einem Zeiger auf den Knoten des generalisierten Arrays, der die Daten dieses Suffixes enthält.

2. **einen Index für den Start des Bereichs mit den ersten drei Zeichen der im generalisierten Suffix Array enthaltenen Suffixe**
   Auch dieser Index besteht aus zwei Teilen: (1) den ersten drei Zeichen des Suffixes, das momentan das lexikografisch kleinste Suffix mit diesen drei Anfangszeichen ist, und (2) einem Zeiger auf den Knoten des generalisierten Arrays, der die Daten dieses Suffixes enthält.

3. **einen Index für das Ende des Bereichs mit den ersten drei Zeichen der im generalisierten Suffix Array enthaltenen Suffixe**
   Dieser Index besteht ebenfalls aus zwei Teilen: (1) den ersten drei Zeichen des Suffixes, das momentan das lexikografisch größte Suffix mit diesen drei Anfangszeichen ist, und (2) einem Zeiger auf den Knoten des generalisierten Arrays, der die Daten dieses Suffixes enthält.

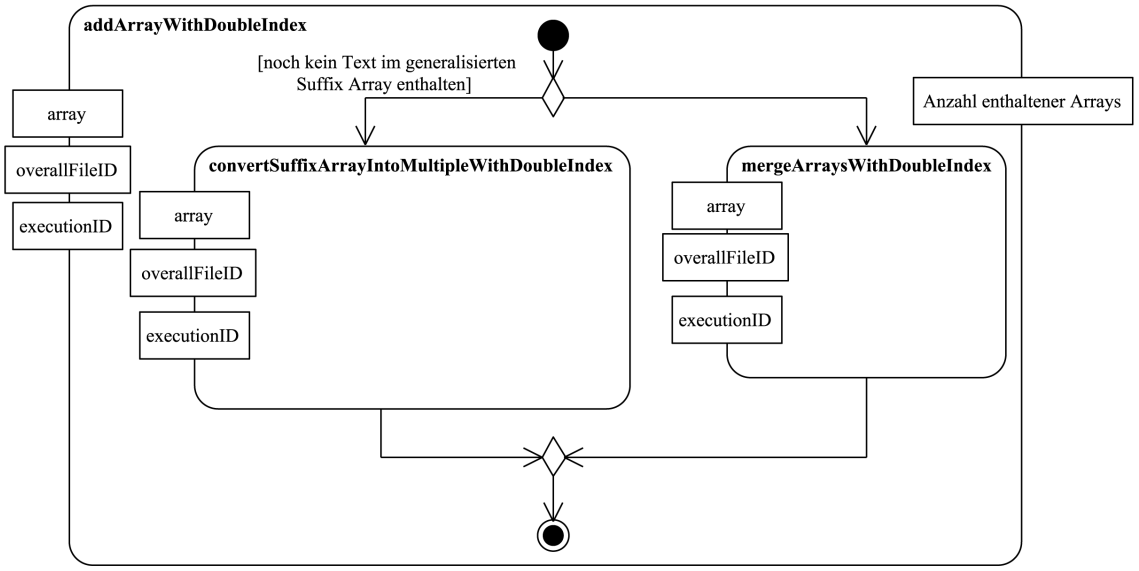


Abbildung 3.25: Ablauf der Operation `addArrayWithDoubleIndex`

Nachdem diese Datenstrukturen erzeugt sind, wird noch eine Datenstruktur für die enthaltenen Features angelegt und die Text-ID des zu konvertierenden Suffix Arrays den Text-IDs der im generalisierten Suffix Array enthaltenen Texte hinzugefügt. Um einen Direktzugriff auf die im generalisierten Suffix Array vorhandenen Texte zu haben, werden diese in einer Datenstruktur namens `dataBuffer` vorgehalten. Dort

werden sie so abgelegt, dass über die Text-ID ein direkter Zugriff auf den eigentlichen Text möglich ist. Dieses Ablegen erfolgt in der Operation `getDataBuffer`, die in Abbildung 3.26 zu sehen ist.

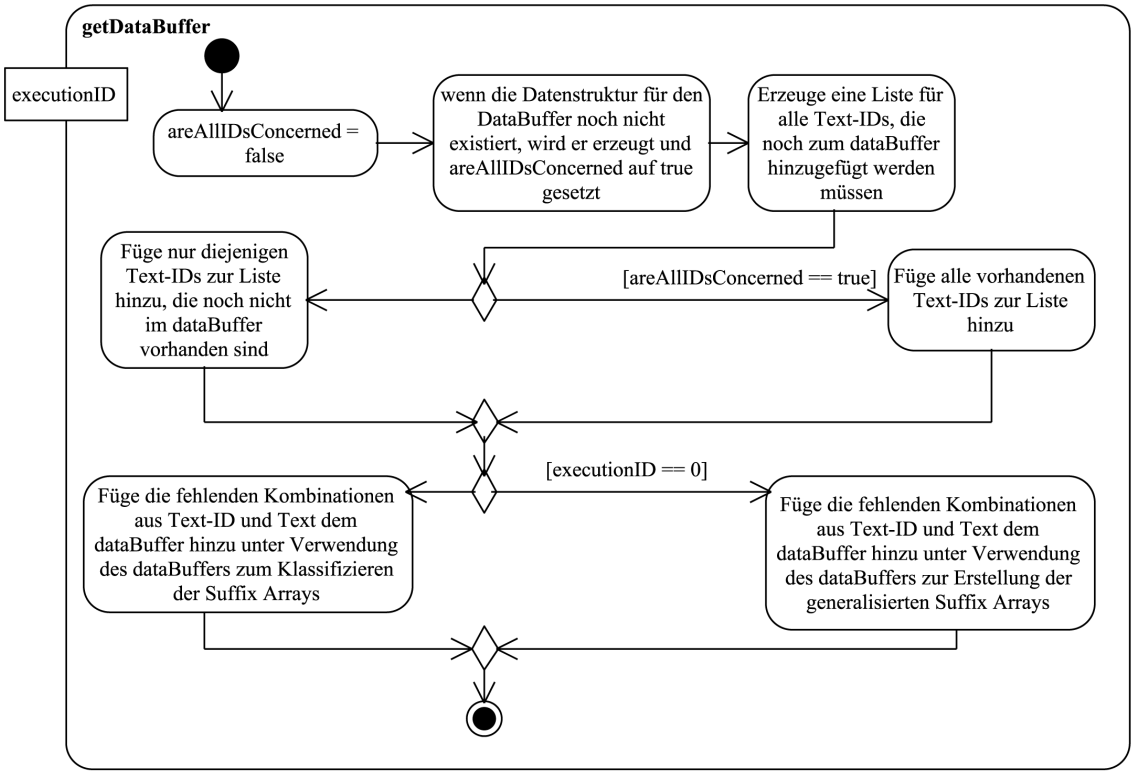


Abbildung 3.26: Ablauf der Operation `getDataBuffer`

Anschließend wird über die einzelnen Elemente des Suffix Arrays iteriert. Das bedeutet, jedes Element des Suffix Arrays für den einzelnen Text wird in ein Element des generalisierten Suffix Arrays umgewandelt. Die Daten dieses Elements werden in einem neuen Knoten gespeichert, der mit der Operation `addNodeWithListIndex` erzeugt wird. Die Operation ist in Abbildung 3.27 auf S. 202 abgebildet. Bis zu diesem Punkt wurde das eigentliche Textsuffix noch nicht betrachtet. Um den Index des generalisierten Suffix Arrays korrekt zu aktualisieren, muss dieser jetzt ausgelesen werden und sein erstes und seine ersten drei Zeichen daraufhin untersucht werden, ob sie bereits in den entsprechenden Indizes enthalten sind. Sind sie enthalten, so wird nur der Zeiger für das Ende des Bereichs der ersten drei Zeichen auf den neuen Knoten gesetzt. Das genügt, da das Suffix Array für den einzelnen Text lexikografisch sortiert ist und lediglich eine Konvertierung vorgenommen wird. Das bedeutet, an der Reihenfolge der Suffixe wird keine Änderung vorgenommen, deshalb ist die einzige nötige Aktualisierung die des Endezeigers des Bereichs der drei ersten Zei-

chen des Suffixes. Sind die Zeichen dagegen nicht enthalten, müssen sowohl sie als auch der Knoten, der gerade in das generalisierte Suffix Array eingefügt wird, in den entsprechenden Indizes ergänzt werden.

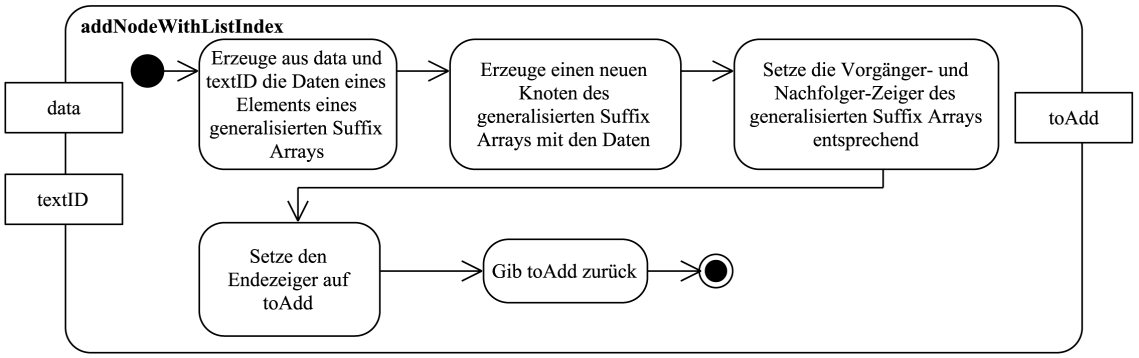


Abbildung 3.27: Ablauf der Operation `addNodeWithListIndex`

Nachdem alle Elemente des einzelnen Suffix Arrays konvertiert wurden, ist die Operation beendet und ein generalisiertes Suffix Array mit einem Text und den beschriebenen Indexstrukturen entstanden. Einen Überblick über die gesamte Operation liefert Abbildung 3.28 auf S. 203.

Ist in einem generalisierten Suffix Array bereits mindestens ein Text enthalten und soll ein weiterer hinzugefügt werden, so erfolgt das mit der Operation `mergeArrays-WithDoubleIndex`. Sie ist in der Abbildung 3.29 auf S. 207 zu sehen. Innerhalb der Operation wird zunächst die ID des hinzuzufügenden Textes in der Datenstruktur für die im generalisierten Suffix Array vorhandenen Texte ergänzt. Dann erfolgt das Hinzufügen des eigentlichen Textes mit der Operation `getDataBuffer`, die zuvor bereits beschrieben wurde.

Das Prinzip des Verschmelzens des generalisierten Suffix Arrays mit dem Suffix Array des einzelnen Textes beruht darauf, dass das Suffix Array linear durchlaufen wird und jeweils das nächste Element in das generalisierte Suffix Array eingefügt wird. Dieses Einfügen muss so erfolgen, dass die lexikografische Sortierung aufrecht erhalten wird. Um das generalisierte Suffix Array nicht auch linear durchlaufen zu müssen, wie im Ursprungsalgorithmus vorgeschlagen, wird innerhalb der Implementierung auf die Indexstrukturen zurückgegriffen. Durch diese kann der Bereich, in den das aktuell einzufügende Suffix eingefügt werden muss, gefunden werden, ohne davorliegende Bereiche durchlaufen zu müssen.

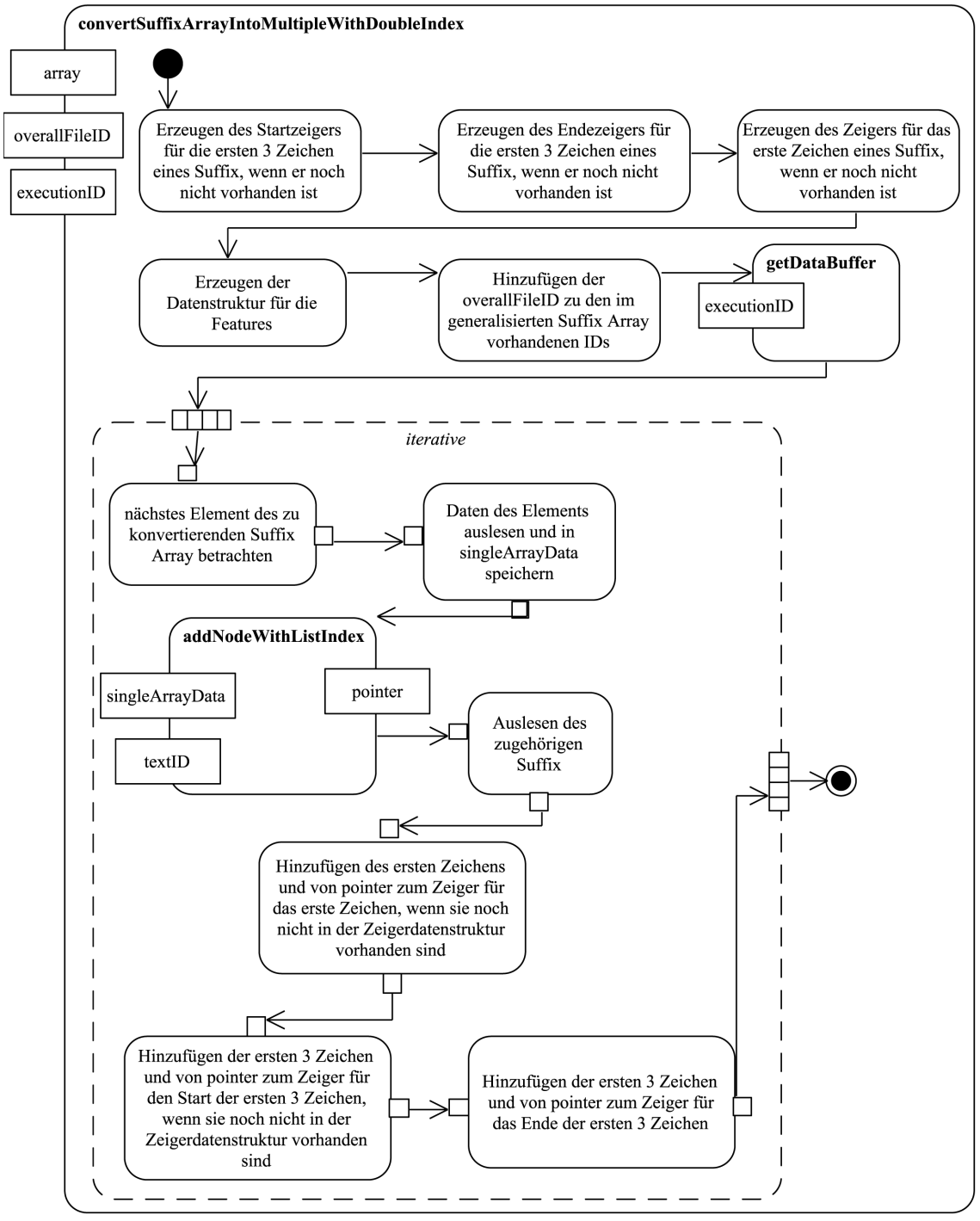


Abbildung 3.28: Ablauf der Operation `convertSuffixArrayIntoMultipleWithDoubleIndex`

Durch dieses „Springen“ innerhalb der Datenstruktur des generalisierten Suffix Arrays sind für die Schleife, mit der der Durchlauf über die beiden Suffix Arrays durchgeführt wird, drei Bedingungen nötig: eine Variable `stopLoop`, die einen sofortigen

Abbruch der Schleife bewirkt, darf nicht `true` sein und beide Zeiger auf den aktuell betrachteten Knoten der jeweiligen Liste - des generalisierten Suffix Arrays und des Suffix Arrays für den einzelnen Text - dürfen nicht ins Leere zeigen, also nicht gleich `null` sein. Ist eine der Bedingungen nicht erfüllt, so wird die Schleife abgebrochen. Innerhalb der Schleife wird als Erstes überprüft, ob sich der Iterator über die Knoten des Suffix Arrays für den einzelnen Text bewegt hat. Ist das der Fall, so wurde im Schritt zuvor das vorangegangene Suffix in das generalisierte Suffix Array eingefügt. Das bedeutet, zunächst müssen die Daten des jetzt einzufügenden Suffixes ausgelesen werden und aus ihnen ein generalisierter Knoten erstellt werden. Zusätzlich werden das erste Zeichen und die ersten drei Zeichen des einzufügenden Suffixes bestimmt, um alle Vergleiche mit den Indexstrukturen durchführen zu können. Eine Verkürzung der Suche nach der Einfügeposition ergibt sich, wenn das erste Zeichen des neu einzufügenden Suffixes mit dem ersten Zeichen des zuletzt eingefügten Suffixes übereinstimmt.[1]
Es können also zwei Fälle unterschieden werden:

1. **Das erste Zeichen des einzufügenden Suffixes unterscheidet sich vom letzten eingefügten ersten Zeichen.**
   In diesem Fall wird aus der Indexstruktur für das erste Zeichen der Zeiger auf den Knoten des generalisierten Suffix Arrays ausgelesen, der auf das erste Zeichen des einzufügenden Suffixes zeigt. Hier lassen sich für die nächsten Schritte wiederum zwei Fälle unterscheiden:

   a) **Es existiert ein Knoten, auf den dieser Zeiger zeigt.**
      Dieser Knoten bildet den Beginn des Einfügebereichs für das einzufügende Suffix, daher wird der Zeiger für den aktuellen Knoten, also den Knoten, ab dem Zeichenkettenvergleiche durchgeführt werden könnten, des generalisierten Suffix Arrays auf diesen Knoten gesetzt. Außerdem wird geprüft, ob sich nicht nur das erste Zeichen des einzufügenden Suffixes bereits im generalisierten Suffix Array befindet, sondern sich dort sogar die ersten drei Zeichen befinden, da dann der Einfügebereich weiter eingeschränkt würde. Je nachdem, ob das der Fall ist oder nicht, werden entsprechende `boolean` Variablen mit entsprechenden Werten versehen.

   b) **Es existiert kein Knoten, auf den dieser Zeiger zeigt.**
      Da es keinen Knoten mit dem ersten Zeichen des einzufügenden Suffixes gibt, müsste man, ausgehend von der letzten Einfügeposition, jedes

1 Zur Erinnerung: Es werden nur Knoten aus dem Suffix Array des einzelnen Textes eingefügt. Das heißt, wenn die beiden genannten Zeichen übereinstimmen, befindet man sich bereits im richtigen Bereich des generalisierten Suffix Arrays.

Suffix des generalisierten Suffix Arrays mit dem einzufügenden Suffix vergleichen, bis man eines findet, das größer ist. Durch den vorhandenen Index und die lexikografische Sortierung kann man jedoch das Suffix finden, das mit dem nächstgrößeren, im generalisierten Suffix Array vorhandenen Zeichen beginnt.[1]

Dafür müssen mehrere Fälle unterschieden werden, da man zwei Sonderzeichen das Endezeichen und das Leerzeichen, sowie den Übergang von Ziffern zu Buchstaben gesondert behandeln muss. Die lexikografische Sortierung ist hier in aufsteigender Reihenfolge: Endezeichen, Leerzeichen, 0 bis 9 und „a“ bis „z“. Hat man das Zeichen bestimmt, welches das nächstgrößere im Index ist, so setzt man den Zeiger für den aktuell zu betrachtenden Knoten im generalisierten Suffix Array auf diesen gefundenen Knoten.

Der Fall ist in der Abbildung 3.30 auf S. 208 zu sehen.

2. **Das erste Zeichen des einzufügenden Suffixes ist gleich dem letzten eingefügten Zeichen.**
   Das bedeutet, der Bereich, in dem die Einfügeposition liegt, ist bereits derjenige, der im generalisierten Suffix Array betrachtet wird. Jetzt können wiederum zwei Fälle unterschieden werden:

   a) **Die ersten drei Zeichen des einzufügenden Suffixes unterscheiden sich von den letzten drei eingefügten Zeichen.**
      Das heißt, der Einfügebereich ist zwar grob der richtige, jedoch ist eine feinere Bereichsbegrenzung noch möglich. Dafür wird zunächst geprüft, ob sich die ersten drei Zeichen des einzufügenden Suffixes bereits im Index für die ersten drei Zeichen befinden. Ist das der Fall, wird die entsprechende Variable mit `true` markiert und der Zeiger für den aktuellen Knoten des generalisierten Suffix Arrays auf den ersten Knoten des feineren Bereichs gesetzt. Im anderen Fall kann der Bereich nicht weiter eingegrenzt werden.

1 Als kleines Beispiel kann man annehmen, dass man ein Suffix einfügen will, welches mit dem Zeichen „e“ beginnt. Zuvor wurde ein Suffix mit Zeichen „b“ eingefügt und im Index ist kein Suffix mit dem Zeichen „e“ enthalten. Nun müsste man jedes Suffix zwischen „b“ und dem Suffix, das mit einem größeren Zeichen, bspw. „g“ beginnt, mit dem einzufügenden Suffix vergleichen. Oder man nutzt aus, dass man das nächstgrößere Zeichen ausgehend von „e“ im Index suchen kann, in diesem Fall „g“, und direkt zum entsprechenden Knoten „springen“ kann.

b) Die ersten drei Zeichen des einzufügenden Suffixes unterscheiden sich nicht von den letzten drei eingefügten Zeichen.
In diesem Fall ist der gerade betrachtete Bereich des generalisierten Suffix Arrays schon der korrekte Einfügebereich. Das bedeutet, es werden nur die entsprechenden Variablen, die das anzeigen, mit den entsprechenden Werten versehen.

Der Fall ist in der Abbildung 3.31 auf S. 208 zu sehen.

Wurde in beiden Fällen kein Knoten des generalisierten Suffix Arrays bestimmt, der den Anfang des Bereichs markiert, in dem sich die Einfügeposition befindet oder vor dem eingefügt werden soll, so kann die Variable `stopLoop` auf `true` gesetzt werden. Denn das bedeutet, dass alle im generalisierten Suffix Array vorhandenen Suffixe kleiner als das einzufügende Suffix sind und demnach die Einfügeposition dieses Suffixes und aller nachfolgenden Suffixe des Suffix Arrays für den einzelnen Text nach dem letzten Element des generalisierten Suffix Arrays liegen. Ist dagegen ein solcher Knoten bestimmt worden, so wird das Suffix dieses Knotens ermittelt, um mit dem einzufügenden Suffix verglichen zu werden. In beiden Fällen wird nun überprüft, ob ein Vergleich der beiden Zeichenketten durchgeführt werden muss oder nicht.

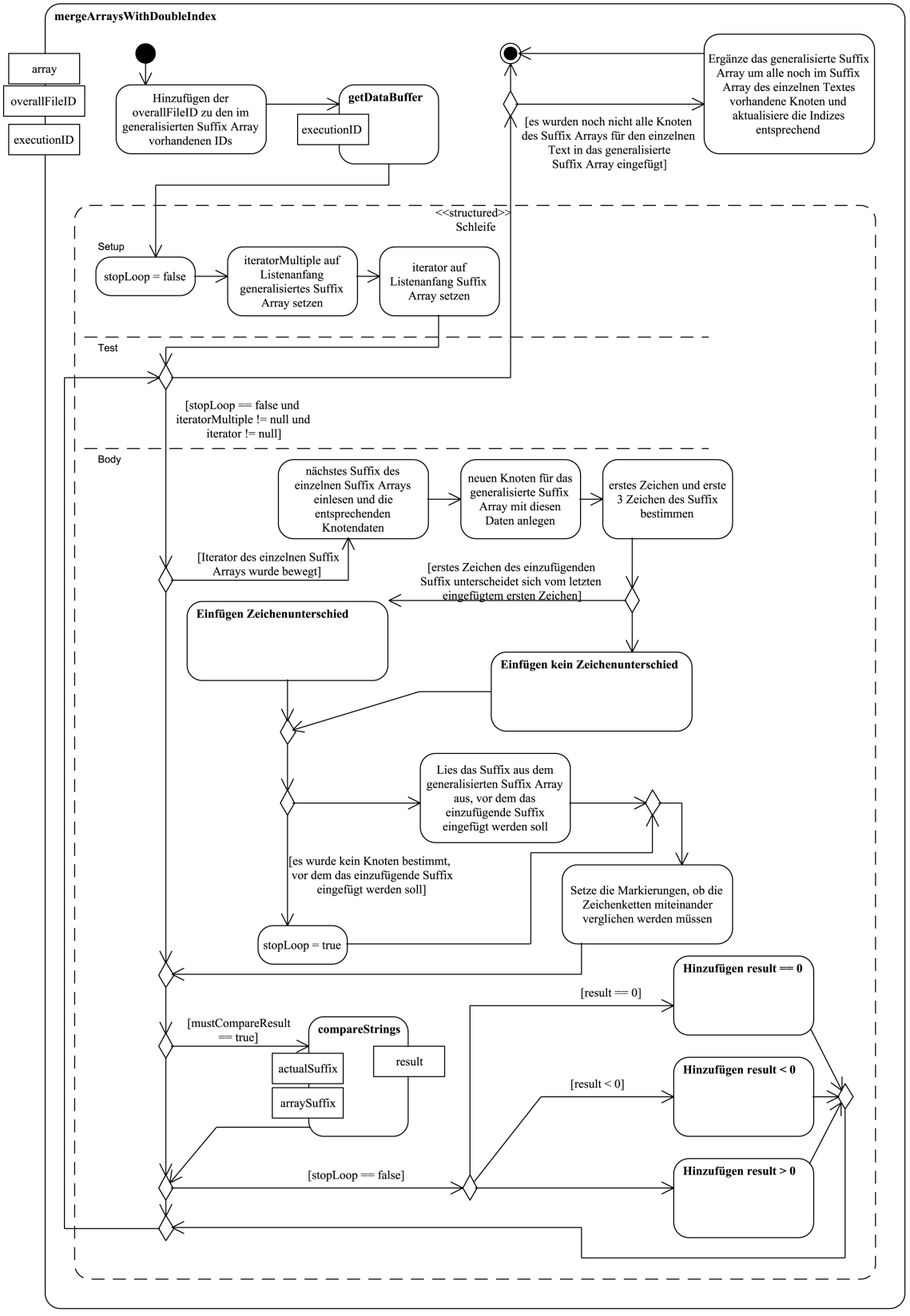


Abbildung 3.29: Ablauf der Operation `mergeArraysWithDoubleIndex`

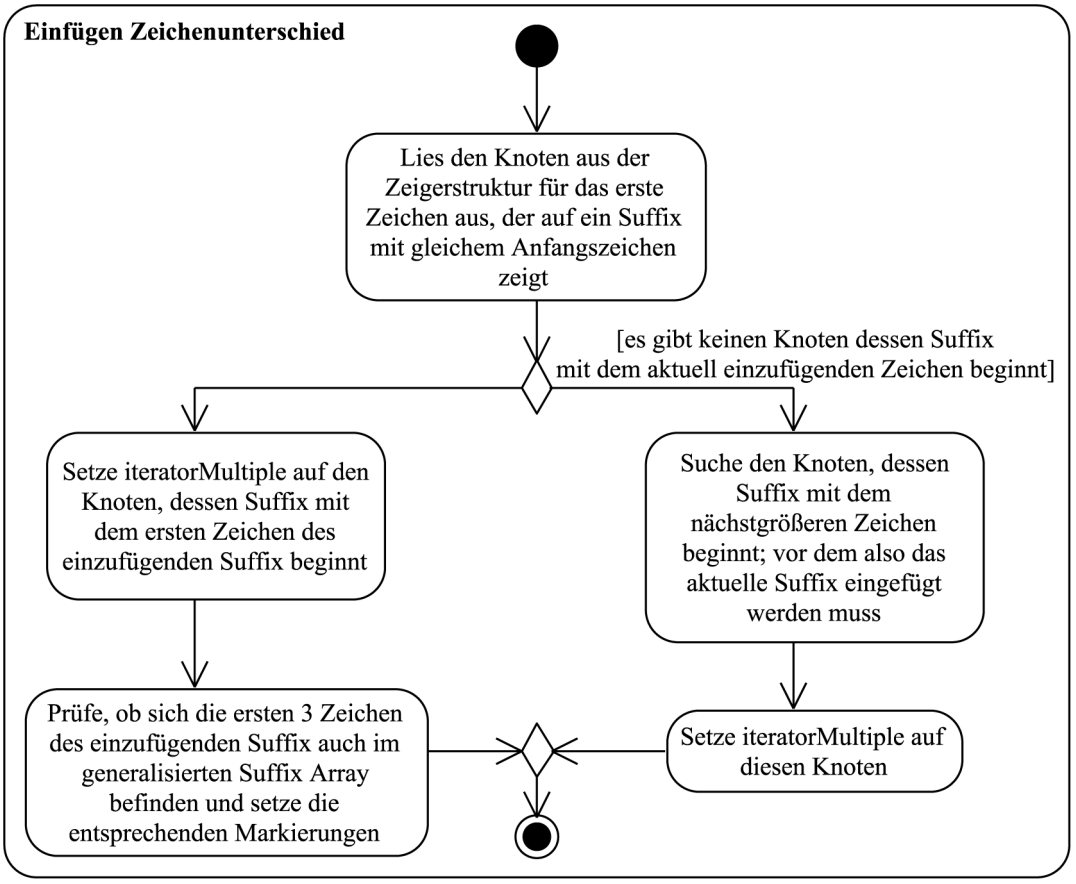


Abbildung 3.30: Ablauf der Aktivität Einfügen Zeichenunterschied

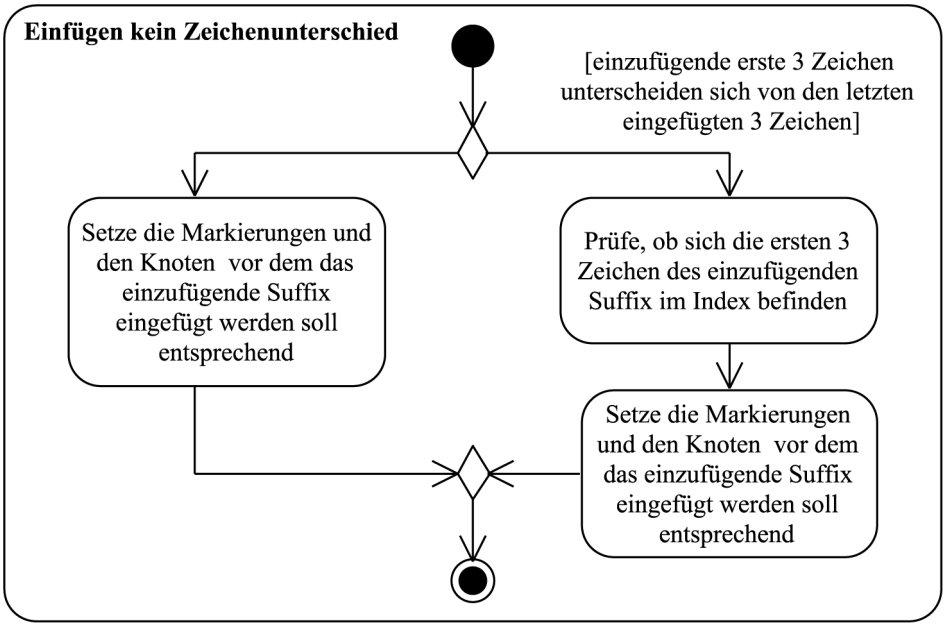


Abbildung 3.31: Ablauf der Aktivität Einfügen kein Zeichenunterschied

Bei der Überprüfung, ob ein Vergleich der beiden Zeichenketten durchgeführt werden muss oder nicht, können drei Fälle unterschieden werden:

1. **Die Verarbeitung soll weitergeführt werden und das erste Zeichen des einzufügenden Suffixes befindet sich im generalisierten Suffix Array.**
   In diesem Fall muss ein lexikografischer Vergleich der beiden Zeichenketten vorgenommen werden, da man zwar den groben Einfügebereich festgelegt hat, jedoch nicht die genaue Einfügeposition.

2. **Die Verarbeitung soll weitergeführt werden.**
   Dadurch, dass das erste Zeichen des einzufügenden Suffixes nicht im Index enthalten ist, weiß man auch, dass die ersten drei Zeichen nicht enthalten sind. Dieser Fall kann nur eintreten, wenn zuvor das nächstgrößere Zeichen im generalisierten Suffix Array gesucht wurde. Das bedeutet gleichzeitig, dass das Suffix des Knotens, auf den der Zeiger für den aktuellen Knoten des generalisierten Suffix Arrays zeigt, in jedem Fall größer ist als das einzufügende Suffix. Das wiederum bedeutet, ein Vergleich der Zeichenketten miteinander ist überflüssig, da das Ergebnis bereits feststeht.

3. **Die Verarbeitung soll nicht weitergeführt werden.**
   Das heißt, dass die Schleife abgebrochen werden soll. Dann ist auch ein Vergleich von Zeichenketten nicht mehr nötig.

Wenn ein Zeichenkettenvergleich stattfinden soll, wird die Operation namens `compareStrings` aufgerufen. Diese Operation vergleicht die zwei übergebenen Zeichenketten, also das einzufügende Suffix und das aktuelle Suffix des generalisierten Suffix Arrays, lexikografisch miteinander. Zu beachten sind dabei zwei Dinge: 1. Die beiden Sonderzeichen - Endezeichen und Leerzeichen - sind kleiner als alle anderen Zeichen. Sie müssen gesondert verglichen werden. 2. Es genügt, die beiden Zeichenketten zeichenweise miteinander zu vergleichen, bis sich der erste Unterschied ergibt. Dann kann der Vergleich abgebrochen werden. Je nach Ergebnis des Vergleichs wird eine Zahl zurückgegeben, die dieses Ergebnis anzeigt: eine Null, wenn beide Zeichenketten gleich sind, eine Eins, wenn die zweite übergebene Zeichenkette, also das einzufügende Suffix, kleiner ist als das Suffix des generalisierten Suffix Arrays, und eine minus Eins im verbleibenden Fall. Als Abbildung 3.32 sieht man den Ablauf auf S. 210.
Steht das Resultat fest und soll die Verarbeitung auch nicht abgebrochen werden, so findet je nachdem, welches Ergebnis sich aus dem Zeichenkettenvergleich ergeben

hat, eine andere Verarbeitung statt. Soll die Verarbeitung abgebrochen werden, so wird keiner der nun beschriebenen Schritte ausgeführt, sondern die Schleife direkt verlassen.

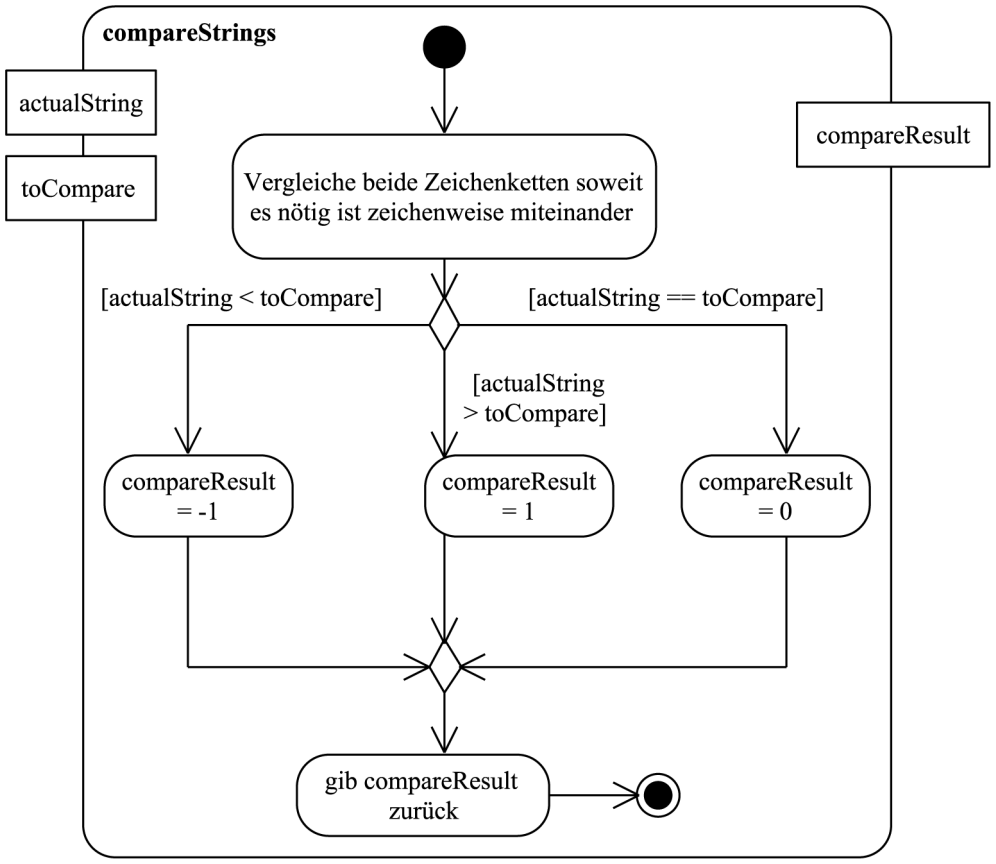


Abbildung 3.32: Ablauf der Operation `compareStrings`

1. **Hinzufügen des Suffixes, wenn es dem Suffix des generalisierten Suffix Arrays entspricht.**
   Sind die beiden Suffixe gleich, so gehören sie in einen Knoten des generalisierten Suffix Arrays. Das bedeutet, die Daten des Suffixes des einzelnen Textes werden den Daten des Suffixes im generalisierten Suffix Array hinzugefügt. Der Knoten des generalisierten Suffix Arrays wird also um eine weitere Kombination aus Text-ID und Suffixposition ergänzt. Handelt es sich um ein Suffix, das länger als zwei Zeichen ist, so ist es im Kontext dieser Arbeit ein *Feature* der Texte, die im generalisierten Suffix Array vorhanden sind.[1] Das bedeutet, das Suffix wird mit den entsprechenden benötigten Informationen der Datenstruktur der Features des generalisierten Suffix Arrays hinzugefügt.

1 Siehe Kapitel 3.2.

Abschließend kann der Zeiger auf das aktuelle Element des Suffix Arrays für den einzelnen Text auf den nächsten Knoten weiterbewegt werden und die entsprechende Variable auf `true` gesetzt werden. Die Abbildung 3.33 zeigt den Ablauf.

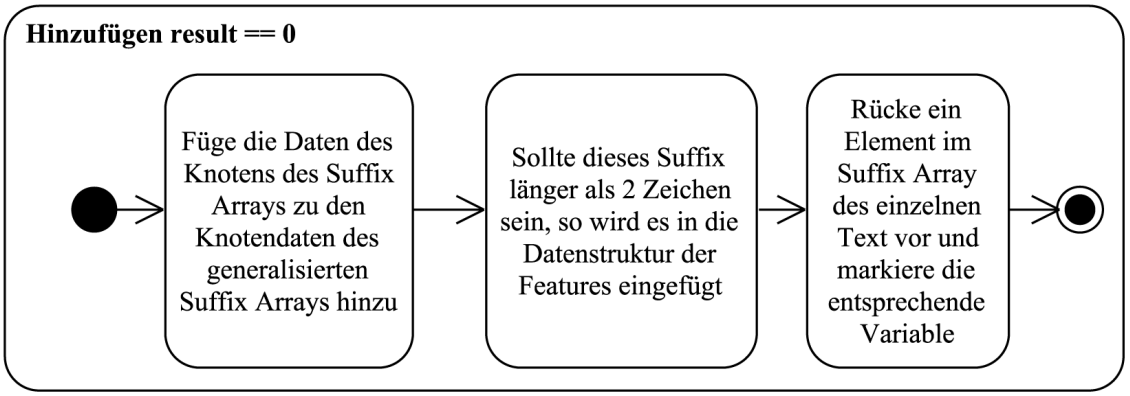


Abbildung 3.33: Ablauf der Aktivität Hinzufügen $result == 0$

2. **Hinzufügen des Suffixes, wenn das Suffix des generalisierten Suffix Arrays kleiner als das einzufügende Suffix ist.**
   In diesem Fall ist der grobe Bereich für das Einfügen gefunden, jedoch noch nicht die genaue Einfügeposition. Um den Bereich weiter verfeinern zu können, wird zunächst geprüft, ob sich die ersten drei Zeichen des einzufügenden Suffixes bereits im generalisierten Suffix Array befinden.

   Ist das so, dann zeigt der Zeiger für den aktuellen Knoten des GSA auf einen Knoten, der ebenfalls in diesem Bereich liegt, jedoch ein kleineres Suffix beinhaltet als das einzufügende. Daher wird der Zeiger für den aktuellen Knoten des GSA auf seinen Nachfolger bewegt, da dieses Suffix größer sein muss als das aktuelle. Existiert dieser Nachfolger, so wird dessen Suffix ermittelt und die Markierung gesetzt, dass wiederum ein Vergleich dieses Suffixes mit dem einzufügenden vorzunehmen ist. Man überspringt dann in der Verarbeitung den ersten Teil des Suchens nach der Einfügeposition und beginnt direkt beim Vergleich der Zeichenketten mit allen nachfolgenden Schritten. Existiert dieser Nachfolger dagegen nicht, so bedeutet das, dass der Zeiger für den aktuellen Knoten des GSA auf den letzten Knoten des generalisierten Suffix Arrays zeigt und die Schleife abgebrochen werden kann.

   Befinden sich die ersten drei Zeichen des einzufügenden Suffixes dagegen nicht im Index, so bedeutet das, dass man den Bereich mit den drei Zeichen des ak-

tuellen Knotens des GSA überspringen kann, da das Suffix des GSA kleiner ist als das einzufügende. Mit Hilfe des Index für das Ende des Bereichs mit diesen drei Zeichen kann man direkt zum letzten Knoten dieses Bereichs springen und den Zeiger für den aktuellen Knoten des GSA auf den Nachfolger dieses letzten Knotens des Bereichs setzen. Dann gilt genau das gleiche, wie oben genannt: Existiert dieser Nachfolger, muss erneut ein Zeichenkettenvergleich vorgenommen werden, ansonsten wird die Schleife abgebrochen.

Die Verarbeitung ist in Abbildung 3.34 dargestellt.

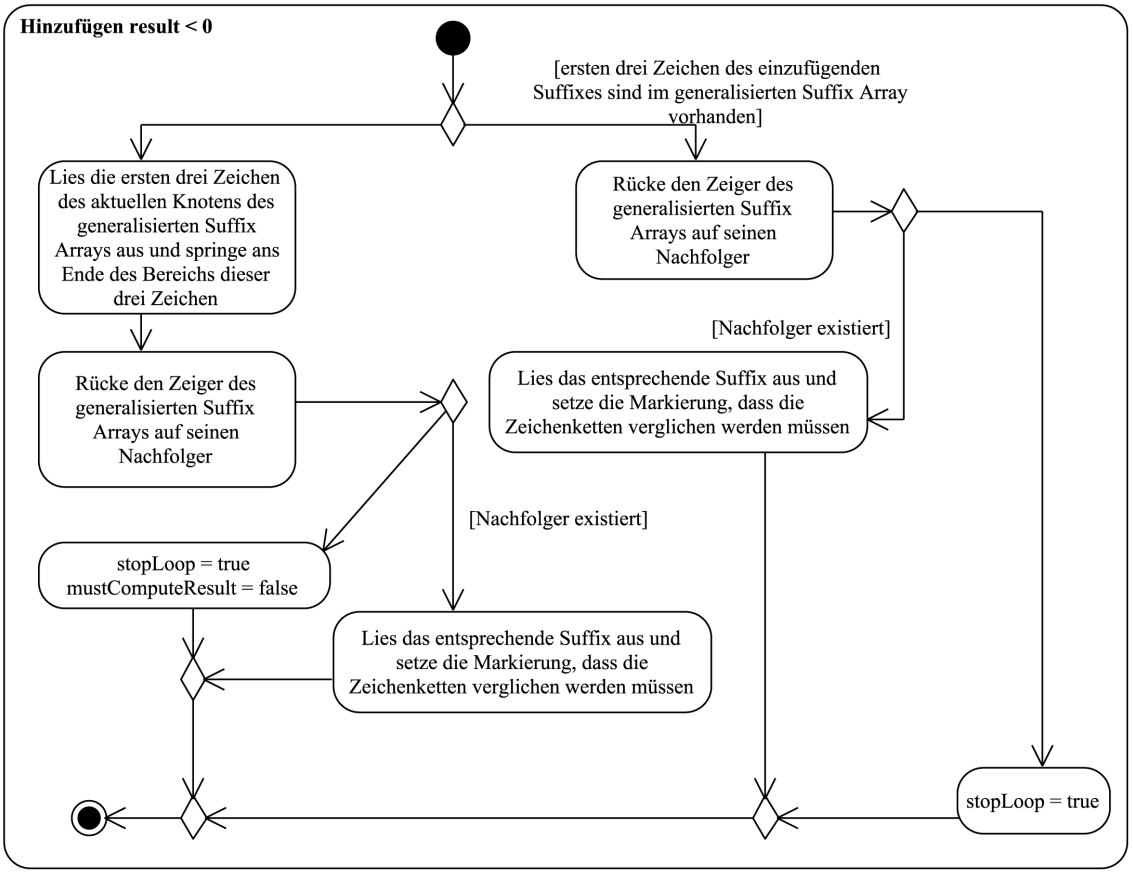


Abbildung 3.34: Ablauf der Aktivität Hinzufügen $result < 0$

3. **Hinzufügen des Suffixes, wenn das Suffix des generalisierten Suffix Arrays größer als das einzufügende Suffix ist.**
   Das bedeutet, der Knoten, vor dem das einzufügende Suffix eingefügt werden soll, ist bereits bestimmt. Man fügt also den umgewandelten Knoten des Suffix Arrays für den einzelnen Text in das generalisierte Suffix Array ein, indem die

Vorgänger- und Nachfolgerzeiger der vorhandenen Knoten entsprechend verändert werden. Abschließend müssen die Indexstrukturen noch aktualisiert werden.

Dafür wird zunächst geprüft, ob die ersten drei Zeichen des eingefügten Suffixes bereits im GSA vorhanden sind. Ist das der Fall, so muss geprüft werden, ob das eingefügte Suffix am Anfang oder am Ende des Bereichs mit den drei Zeichen eingefügt wurde. Dann muss entsprechend der Index für den Anfang oder das Ende aktualisiert werden, so dass der entsprechende Zeiger auf den neu eingefügten Knoten zeigt. Wird der Knoten in der Mitte des Bereichs eingefügt, ergeben sich keine Auswirkungen auf die Indexstrukturen. Außerdem muss überprüft werden, ob es sich bei dem eingefügten Suffix um das lexikografisch kleinste Suffix mit diesem Anfangszeichen im GSA handelt. Dafür wird das entsprechende Suffix mit Hilfe des Index ausgelesen und die beiden Suffixe werden miteinander verglichen. Ist das eingefügte Suffix kleiner als das vorhandene, wird der Index für das erste Zeichen entsprechend geändert.

Sind die ersten drei Zeichen des eingefügten Suffixes noch nicht im GSA enthalten, so wird zunächst überprüft, ob das erste Zeichen bereits vorhanden ist. Ist es noch nicht vorhanden, so werden das Zeichen und der Zeiger auf den gerade eingefügten Knoten im Index ergänzt. Ist es vorhanden, so wird das gleiche Verfahren angewendet, wie gerade beschrieben: Es erfolgt ein Vergleich, um den Index zu ändern, falls das eingefügte Suffix kleiner ist als das bisher kleinste im GSA mit diesem Anfangszeichen. In beiden Fällen wird der Index für die drei Anfangszeichen um diese und den Zeiger auf den eingefügten Knoten ergänzt.

Abschließend wird die Markierung gesetzt, dass ein Knoten des Suffix Arrays des einzelnen Textes eingefügt worden ist und entsprechend der nächste Knoten betrachtet werden kann. Die Abbildung 3.35 auf S. 214 zeigt die Verarbeitung.

Sind an dieser Stelle noch alle Bedingungen erfüllt, so wird die Schleife so lange weitere Male durchlaufen, bis entweder explizit ein Abbruch erfolgt oder die Zeiger für die aktuellen Knoten des GSA und des Suffix Arrays für den einzelnen Text ins Leere zeigen. Wird die Schleife verlassen, kann der Fall eintreten, dass noch nicht alle Knoten des Suffix Arrays für den einzelnen Text dem GSA hinzugefügt wurden, weil die noch verbliebenen Suffixe größer als alle Suffixe des GSA sind. In diesem Fall werden alle noch verbliebenen Knoten ans Ende des GSA angehängt und die Indizes entsprechend aktualisiert. Danach ist das Verschmelzen der beiden Suffix Arrays beendet.

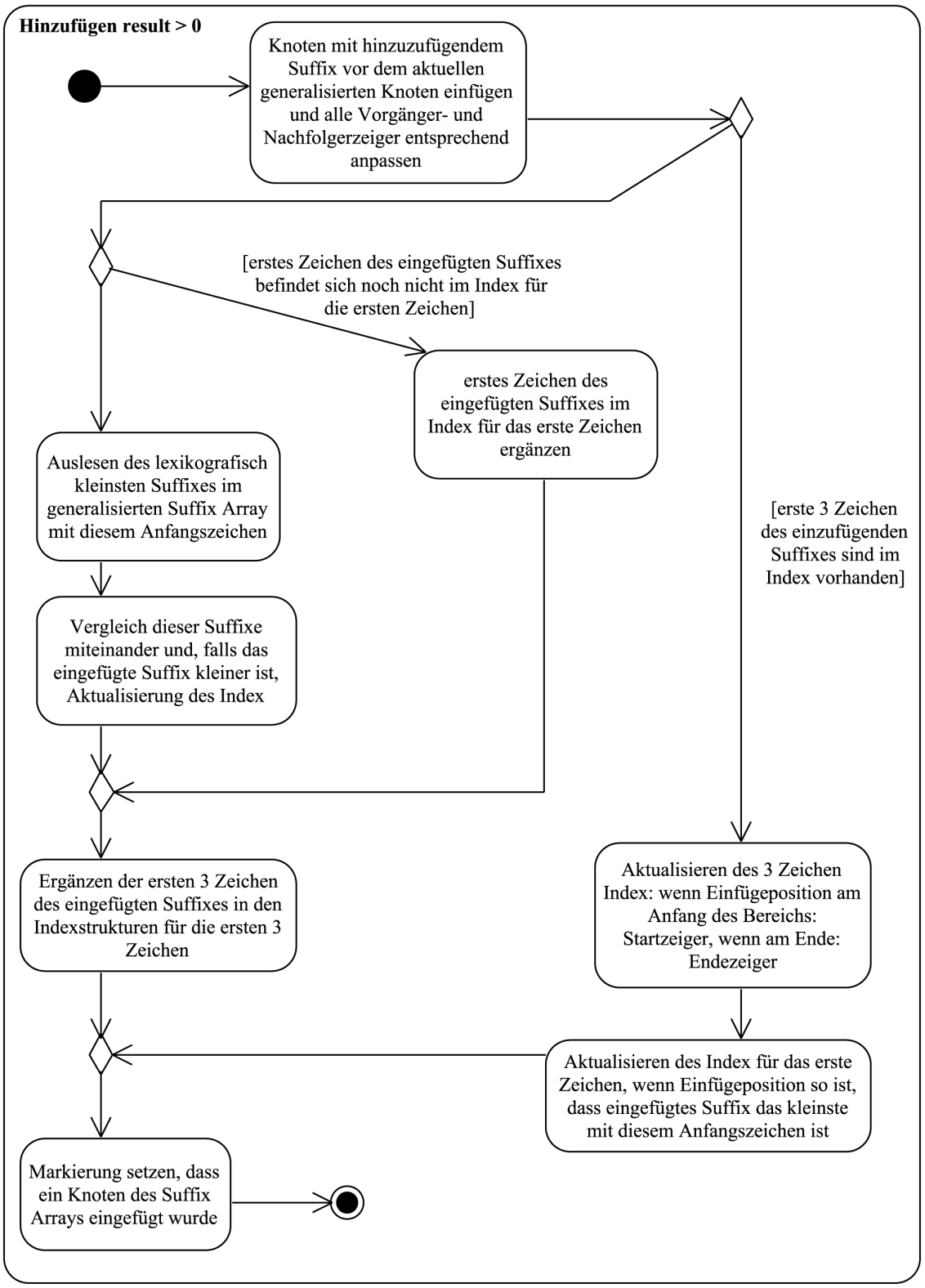


Abbildung 3.35: Ablauf der Aktivität Hinzufügen $result > 0$

3.2 Text-Suffix-Fragment-Feature-Ermittlung

3.2.1 Definition Text-Suffix-Fragment-Feature

In der vorliegenden Arbeit werden zum Klassifizieren und Clustern von natürlichsprachlichen Texten so genannte Text-Suffix-Fragment-Features (TSF-Features) ermittelt.[1] In Definition 6 wurde bereits definiert, was ein Text-Suffix ist. Darauf aufbauend legen die beiden folgenden Definitionen fest, was ein Text-Suffix-Fragment-Feature ist.

Definition 21. *Ein **Text-Suffix-Fragment (TSF)** ist ein Teiltext eines Suffixes eines Textes T.*

Definition 22. *Ein **Text-Suffix-Fragment-Feature** ist ein Text-Suffix-Fragment, das*

- ***mindestens doppelt** im **Text** selbst oder[2] in der **Klasse** des Textes vorkommt und*
- ***mindestens drei Zeichen lang** ist.*

Anhand eines Beispiels lassen sich die Definitionen nachvollziehen. Der betrachtete Text ist „sitzplatz“. Alle Text-Suffixe dieses Textes sind: das leere Suffix, „z“, „tz“, „atz“, „latz“, „platz“, „zplatz“, „tzplatz“, „itzplatz“ und „sitzplatz“. Betrachtet man nun das Text-Suffix „latz“, so lassen sich folgende Text-Suffix-Fragmente bilden: die einzelnen Zeichen „l“, „a“, „t“ und „z“, jeweils zwei aufeinanderfolgende Zeichen „la“, „at“ und „tz“, drei aufeinanderfolgende Zeichen „lat“ und „atz“ und schließlich das ganze Text-Suffix „latz“, das ebenfalls ein Text-Suffix-Fragment ist.
Der Text „sitzplatz“ allein hat keine Text-Suffix-Fragment-Features laut Definition, da keines der Text-Suffix-Fragmente, das mindestens doppelt vorkommt, länger als zwei Zeichen ist. Dazu muss ein weiterer Text hinzugenommen werden: „stehplatz“. Nimmt man an, dass diese beiden Texte zur selben Klasse gehören, so ergeben sich laut Definition die folgenden TSF-Features: „platz“, „latz“ und „atz“.
Die zweite Definition schränkt die Menge der potenziellen Features ein. Das wird in der vorliegenden Arbeit, genau wie in vielen anderen Ansätzen in der Literatur[3],

1 In Kapitel 2 wurde bereits beschrieben, was ein Feature ist und welche Funktion Features im Rahmen des Klassifizierens und Clusterns von natürlichsprachlichen Texten haben.
2 Das „oder“ ist kein exklusives „entweder oder“.
3 Diese Vorgehensweise wird auch *feature selection* genannt, wobei dann zumeist erst alle potenziellen Features ermittelt werden und dann diese Menge eingeschränkt wird, während in der vorliegenden Arbeit direkt eine Einschränkung stattfindet, die aber durch weitere Methoden ergänzt werden kann. Vgl. Manning u.a. (2008), S. 251-258; Weiss u.a. (2005), S. 25-32, 35 f.; Baeza-Yates u.a. (1999), S. 163 f.; Forster (2006), S. 63-67; Jain u.a. (1999), S. 271; Larsen u.a. (1999), S. 17.

vorgenommen, um das Klassifizieren und das Clustern einer potenziell großen Menge an Texten in einer nicht zu großen Zeitspanne durchführen zu können und die Qualität der Ergebnisse zu steigern. Diese Einschränkungen werden aus folgenden Gründen so gewählt:

- Ein mindestens doppeltes Auftreten eines Features in einem Text weist darauf hin, dass es sich bei diesem Feature um einen wichtigen Bestandteil des Textes handelt. Das gleiche gilt für das doppelte Auftreten eines Features innerhalb einer Klasse.
- Die Beschränkung der Features auf eine minimale Länge von drei Zeichen dient dazu, wichtige, in einem Text verwendete Abkürzungen in die Menge der Features aufzunehmen, aber nicht zu kurze Textfragmente, wie bspw. einzelne Zeichen, zu betrachten.

Selbstverständlich fallen auch bei dem Ansatz der Verfasserin zur Definition der Features evtl. signifikante Features weg. Signifikant heißt in diesem Zusammenhang, dass es sich um Features handelt, die eine Abgrenzung zwischen zwei oder mehr möglichen Klassen für den betrachteten Text ermöglichen. Das können auch Features sein, die nur einmal in einem Text oder einer Klasse auftauchen, aber so prägnant sind, dass sie direkt die Klasse des Textes festlegen würden. Solche Features werden mit dem hier vorgestellten Ansatz zur Featureermittlung nicht gefunden. Da in der Literatur jedoch auch zu einem großen Teil eine Häufigkeitsbetrachtung der Features stattfindet und solche, die zu selten oder zu oft vorkommen, ebenfalls nicht betrachtet werden, stellt das für die angedachten Experimente keinen Nachteil dar.[1] Die Einschränkung ist zudem nötig, um den Umfang der Featuremenge nicht zu groß werden zu lassen, damit die Klassifizierungs- und Clusteralgorithmen noch rechenbar bleiben.

3.2.2 Algorithmen zur Ermittlung der Text-Suffix-Fragment-Features

3.2.2.1 Algorithmus zur Ermittlung der Text-Suffix-Fragment-Features zur Klassifizierung eines Textes

Das Klassifizieren eines Textes setzt voraus, dass Trainingsdaten vorhanden sind. Für diese werden Features bestimmt, die die Klassen der Trainingsdaten voneinander abgrenzen und die in einem zu klassifizierenden Text gesucht werden können. Der Text kann dann durch die Anwendung von Algorithmen zum Klassifizieren aufgrund der

1 Vgl. Manning u.a. (2008), S. 257; Weiss u.a. (2005), S. 27-30; Forster (2006), S. 63, 65 f.; Larsen u.a. (1999), S. 17.

in ihm vorhandenen Features klassifiziert werden. An dieser Stelle wird beschrieben, wie die TSF-Features für die Texte der Trainingsdaten für alle vorliegenden Klassen ermittelt werden.
Die zu suchenden TSF-Features haben laut Definition zwei Eigenschaften:

1. Sie sind mindestens drei Zeichen lang und
2. sie kommen mindestens doppelt im Text selbst oder in der Klasse der gerade betrachteten Texte vor.

Die Ausgangssituation für das Ermitteln der TSF-Features für die Texte einer Klasse ist das Vorliegen eines generalisierten Suffix Arrays für diese Klasse, das alle zu der Klasse gehörenden Texte enthält. Das bedeutet, als Voraussetzung müssen zunächst alle Texte als Suffix Array vorliegen, um dann, wie in Kapitel 3.1.5.2 beschrieben, zu einem generalisierten Suffix Array verschmolzen zu werden. Ist das geschehen, so können die TSF-Features dieser Klasse ermittelt werden. Durch die Eigenschaften des generalisierten Suffix Arrays wird die Aufgabe, diese TSF-Features zu finden, erleichtert.
Die aufsteigende lexikografische Sortierung stellt sicher, dass sich ähnliche Suffixe im gleichen Bereich des generalisierten Suffix Arrays befinden. Mit ähnlichen Suffixen ist in der vorliegenden Arbeit gemeint, dass es sich um Suffixe handelt, die mit dem gleichen Zeichen oder mit gleichen Teilzeichenketten beginnen. Findet man nun Suffixe, die mit der gleichen mindestens drei Zeichen langen Teilzeichenkette beginnen, so müssen diese Suffixe nur auf die Teilzeichenkette beschränkt werden, die gleich ist, und es handelt sich um ein TSF-Feature. Durch das Vorhandensein des *supra-index*, also des Drei-Zeichen-Index für das generalisierte Suffix Array, lassen sich alle Bereiche finden, in denen die enthaltenen Suffixe mindestens drei gemeinsame Zeichen am Beginn des Suffix aufweisen. Anstatt nun das gesamte Array zu durchlaufen, wird stattdessen der Drei-Zeichen-Index durchlaufen und direkt überprüft, ob der durch den Anfang und das Ende festgelegte Bereich mehr als ein Suffix enthält. Ist das der Fall, so können alle TSF-Features dieses Bereichs ermittelt werden, indem er durchlaufen wird und jeweils die Länge der gemeinsamen Teilzeichenkette zwischen nebeneinanderstehenden Suffixen dieses Bereichs bestimmt wird. Anschließend müssen die Suffixe dann auf die entsprechende Länge gekürzt werden und die Ermittlung der TSF-Features für diesen Bereich des generalisierten Suffix Arrays ist abgeschlossen.
Umfasst der betrachtete Bereich dagegen nur ein Element des GSA, sind also Anfang und Ende des Bereichs gleich, so ist die Bedingung verletzt, dass das Text-Suffix-Fragment mindestens doppelt im Text oder in der Klasse vorkommen muss. Es gibt

aber Fälle, in denen die Bedingung trotzdem erfüllt ist: Das Element des generalisierten Suffix Arrays kann eines sein, in dem gleiche Suffixe mindestens zweier Texte gespeichert sind. Trifft dieser Fall zu, so wird das gesamte Suffix als TSF-Feature ermittelt.
Als Algorithmus ergibt sich der folgende:

Algorithmus 22 ermittleTSF-FeaturesZumKlassifizieren

Eingabe: generalisiertes Suffix Array GSA für eine Klasse von Texten
Ausgabe: Menge der TSF-Features für die Klasse von Texten

```
1: Lies den Drei-Zeichen-Index von GSA aus, das ist: index
2: for all Elemente in index do
3:     Lies den Start des Bereichs aus, das ist: start
4:     Lies das Ende des Bereichs aus, das ist: ende
5:     Lies das Suffix in start aus, das ist: sufStart
6:     if start ≠ ende then
7:         Lies den Nachfolger von start aus, das ist: nachfolger
8:         Lies das Suffix in nachfolger aus, das ist: sufNachfolger
9:         while start ≠ ende do
10:            Zähle gleiche Zeichen am Anfang von sufStart und
               sufNachfolger, das ist: result
11:            Bilde das TSF-Feature als Teilzeichenkette von sufStart[0..result]
               und speichere es
12:            start = nachfolger
13:            sufStart = sufNachfolger
14:            nachfolger auf Nachfolger von nachfolger setzen
15:            Lies das Suffix in nachfolger aus, das ist: sufNachfolger
16:        end while
17:    else
18:        Prüfe, ob start mindestens zwei Text-IDs beinhaltet
19:        if mindestens zwei Text-IDs vorhanden then
20:            Speichere sufStart als TSF-Feature
21:        end if
22:    end if
23: end for
```

Für die beiden Texte „sitzplatz“ und „stehplatz“ ergibt sich das generalisierte Suffix Array, das auf S. 185 zu sehen ist. Der Drei-Zeichen-Index dieses generalisierte Suffix Array wird im Folgenden als Tabelle dargestellt, wobei die Start- und Endelemen-

te nicht durch ihren Index im generalisierten Suffix Array definiert sind, sondern stattdessen die entsprechende Text-ID und Suffix-Position aufgeführt werden:

Teilzeichenkette	Start	Ende
atz	1,7; 2,7	1,7; 2,7
ehp	2,3	2,3
hpl	2,4	2,4
itz	1,2	1,2
lat	1,6; 2,6	1,6; 2,6
pla	1,5; 2,5	1,5; 2,5
sit	1,1	1,1
ste	2,1	2,1
teh	2,2	2,2
tz$	1,8; 2,8	1,8; 2,8
tzp	1,3	1,3
zpl	1,4	1,4

Wie man anhand des Index direkt feststellen kann, umfasst kein Bereich des Drei-Zeichen-Index mehr als ein Element, da alle Start- und Endknoten jeweils gleich sind. Das bedeutet, im Algorithmus wird immer der `else`-Zweig ausgeführt.
erstes Element von `index` auslesen: `atz`
`start` $= 1,7;2,7$
`ende` $= 1,7;2,7$
`sufStart` = `atz$`
`start` == `ende` $\Rightarrow$

Prüfen, ob zwei Text-IDs vorhanden: Das ist der Fall $\Rightarrow$

Speichere das TSF-Feature „atz$“

Diese Verarbeitung wird für alle Elemente des GSA wiederholt, wobei sich noch die TSF-Features „latz$“, „platz$“ und „tz$“ ergeben.

3.2.2.2 Algorithmus zur Ermittlung der Text-Suffix-Fragment-Features zum Clustern eines Textes („AllDoks“)

Im Gegensatz zum Klassifizieren eines Textes müssen für das Clustern keine Trainingsdaten vorhanden sein. Stattdessen werden alle zu clusternden Texte direkt betrachtet, also ihre TSF-Features ermittelt. Mit diesen TSF-Features und den Clusteralgorithmen kann dann ein Clustering erstellt werden. Innerhalb der vorliegenden

Arbeit werden zwei Ansätze für das Ermitteln von TSF-Features für das Clustern vorgestellt. Der erste in diesem Unterkapitel beschriebene Ansatz betrachtet zur Ermittlung der TSF-Features der zu clusternden Texte alle vorliegenden Texte.
Das bedeutet, alle Texte, die einzeln als Suffix Array vorliegen müssen, werden in einem gemeinsamen GSA zusammengeführt. Man betrachtet die Texte also zunächst so, als würden sie zu einer einzigen Klasse gehören. Durch die Erzeugung des GSA aller Texte lassen sich dann die TSF-Features für alle vorliegenden Texte ermitteln. Daher erhält dieser Ansatz den Namen „AllDoks“. Das Ermitteln der TSF-Features erfolgt mit dem gleichen Algorithmus, wie in Kapitel 3.2.2.1 beschrieben. Der dortige Algorithmus 22 muss nur am Anfang um den Schritt der Erzeugung eines gemeinsamen GSA für alle zu clusternden Texte ergänzt werden.
Der Ablauf des Algorithmus anhand der Beispieltexte „sitzplatz“ und „stehplatz“ ist der gleiche, wie zuvor bei der Ermittlung von TSF-Features für die Klassifizierung von Texten. Der einzige Zusatzschritt ist die Erzeugung des GSA für die beiden Texte, der aber im entsprechenden Kapitel, siehe Kapitel 3.1.5.2, nachgelesen werden kann.

3.2.2.3 Algorithmus zur Ermittlung der Text-Suffix-Fragment-Features zum Clustern eines Textes („SingleDoks“)

Auch hier geht es darum, TSF-Features zu ermitteln, um einen Text zu clustern. Im Gegensatz zu dem im vorherigen Abschnitt beschriebenen Ansatz wird aber ein anderer Ansatz zur Bestimmung der TSF-Features gewählt. Beim im Folgenden beschriebenen Ansatz werden die TSF-Features für *einen* Text ermittelt, im Gegensatz zum Ansatz vorher, bei dem die TSF-Features über *alle* zu clusternden Texte ermittelt wurden. Daher erhält der hier beschriebene Ansatz den Namen „SingleDoks“.
Die TSF-Features sind so definiert, dass sie auch in einem einzelnen Text ermittelt werden können, nämlich wenn in diesem Text Text-Suffix-Fragmente doppelt vorkommen, die mindestens drei Zeichen lang sind. Ausnutzen lässt sich auch in diesem Fall wieder die aufsteigend lexikografische Sortierung im Suffix Array des Textes, da hier ebenfalls gleich beginnende Suffixe im gleichen Bereich zu finden sind. Das bedeutet, das Suffix Array kann entweder über einen vorhandenen *supra-index* oder aber auch ohne Index durchlaufen werden und jeweils im Suffix Array benachbarte Suffixe werden daraufhin untersucht, ob sie mit der gleichen Teilzeichenkette beginnen, die mindestens drei Zeichen lang ist. Als Algorithmus ergibt sich also der gleiche wie Algorithmus 22, nur dass der Algorithmus nicht auf dem GSA ausgeführt wird, sondern auf dem Suffix Array für einen einzelnen Text.

Ist dort kein Index vorhanden, so wird nicht über den Index iteriert, sondern direkt über die Elemente des Suffix Arrays. Der Fall, dass ein einzelnes Element des Suffix Arrays ein TSF-Feature des Textes enthält, kann dann aber nicht auftreten.
Für die beiden Texte „sitzplatz“ und „stehplatz“ können einzeln keine TSF-Features ermittelt werden, wenn die Texte geclustert werden sollen.[1] Aus diesem Grund wird hier der Text „abrakadabra“ gewählt. Das Suffix Array für diesen Text sieht wie folgt aus:

1	2	3	4	5	6	7	8	9	10	11	12
12	11	8	1	6	4	9	2	7	5	10	3

Der entsprechende Drei-Zeichen-Index ist der folgende:

Teilzeichenkette	Start	Ende
abr	1,8	1,1
ada	1,6	1,6
aka	1,4	1,4
bra	1,9	1,2
dab	1,7	1,7
kad	1,5	1,5
ra$	1,10	1,10
rak	1,3	1,3

Wie man anhand des Index direkt feststellen kann, umfassen zwei Bereiche mehr als ein Element. Das sind die einzigen beiden Stellen, an denen sich für den einzelnen Text TSF-Features ermitteln lassen.
erstes Element von `index` auslesen: `abr`
`start` $= 1,8$
`ende` $= 1,1$
`sufStart` $=$ `abra$`
`start` $\neq$ `ende` $\Rightarrow$
    `nachfolger` $= 1,1$
`sufNachfolger` $=$ `abrakadabra$`
`result` $= 4$

1 Das liegt an der Definition der TSF-Features. Sollen TSF-Features für jeden der beiden Beispieltexte einzeln bestimmt werden, so sind TSF-Features Text-Suffix-Fragmente, die doppelt in einem der beiden Texte vorkommen. In den verwendeten Beispieltexten existieren solche doppelten Text-Suffix-Fragmente in einem der beiden Texte nicht.

Speichere das TSF-Feature „abra“
`start` $= 1, 1$
`sufStart` = „abrakadabra$“
`nachfolger` $= 1, 6$
`start` = `ende` $\Rightarrow$
    Schleife beenden und nächstes Element von `index` auslesen
Diese Verarbeitung wird für alle Elemente des Index wiederholt, wobei sich noch das TSF-Feature „bra“ ergibt.

3.2.3 Implementierung der Ermittlung von Text-Suffix-Fragment-Features

3.2.3.1 Implementierung der Ermittlung von Text-Suffix-Fragment-Features zur Klassifizierung eines Textes

Um TSF-Features, so wie sie definiert sind, siehe Definition 22 auf S. 215, zu ermitteln und für das Klassifizieren eines Textes oder mehrerer Texte zu verwenden, muss für alle Texte der Trainingsdaten und für alle zu klassifizierenden Texte jeweils ein Suffix Array erstellt werden. Das erfolgt mit der in Kapitel 3.1.4.3 als Algorithmus und in Kapitel 3.1.4.4 als Implementierung vorgestellten Methode. Nach der Erstellung ist der Ablauf der gleiche, wie bereits im Algorithmus-Kapitel, siehe Kapitel 3.2.2.1, beschrieben. Der Ablauf - Erstellen eines generalisierten Suffix Arrays für alle Texte der Trainingsdaten einer Klasse und Ermitteln der TSF-Features über alle Texte der Trainingsdaten dieser Klasse - bleibt der gleiche. Das bedeutet, nachdem ein GSA, wie in Kapitel 3.1.5.2 und Kapitel 3.1.5.3 beschrieben, erstellt wurde, werden die Features der Texte der Trainingsdaten dieser Klasse durch den Aufruf der Operation `extractFeatures`, zu sehen in Abbildung 3.36 auf S. 224, ermittelt.
In der Operation wird zunächst eine Datenstruktur für die Features angelegt. Diese Datenstruktur speichert das TSF-Feature als Zeichenkette und die dazugehörigen IDs der Texte der Trainingsdaten, in denen das entsprechende TSF-Feature vorkommt. Um die Zeichenketten verarbeiten zu können, wird die Operation `getDataBuffer`, die ab S. 200 beschrieben wurde, aufgerufen. Durch sie wird eine Datenstruktur mit den Text-IDs und den Texten der Trainingsdaten gefüllt. Innerhalb einer Schleife werden dann nach und nach alle Inhalte des Drei-Zeichen-Index des GSA verarbeitet. Solange dieser noch unverarbeitete Elemente aufweist, werden das nächste Element sowie der Start- und der Endknoten der drei Zeichen im gerade zu verarbeitenden GSA ausgelesen.

Entsprechen sich die beiden Knoten, so zeigen die jeweiligen Zeiger auf das gleiche Element des GSA und es handelt sich laut Definition 22 nicht um ein TSF-Feature.[1] Dann erfolgt keine weitere Verarbeitung dieses Bereichs des GSA.
Weisen die Zeiger dagegen auf zwei unterschiedliche Knoten, so existiert mindestens ein Paar von Suffixen, aus denen ein TSF-Feature ermittelt werden kann, da sie mit den gleichen drei Zeichen beginnen. Daher wird zunächst der Nachfolger des Startknotens ermittelt und die Anzahl der gemeinsamen Anfangszeichen dieser beiden Suffixe bestimmt. Das erfolgt in der Operation `countEqualBeginningCharacters`, deren Rückgabevariable `result` so lange um eins erhöht wird, wie sich die Anfangszeichen beider Suffixe entsprechen. Sobald die Zeichen nicht mehr gleich sind, endet die Operation und gibt die Anzahl der übereinstimmenden Zeichen zurück. Um jetzt das eigentliche TSF-Feature als solches zu bestimmen, wird eine Teilzeichenkette von Position Null bis `result` eines der beiden Suffixe gebildet und in der Datenstruktur für die Features gespeichert. Zusätzlich werden die Text-IDs der beteiligten Suffixe ermittelt und ebenfalls dort gespeichert. Abschließend wird der Startknoten auf seinen Nachfolger und der Nachfolger auf wiederum seinen Nachfolger gesetzt, um mit der Verarbeitung des Bereichs fortfahren zu können. Das erfolgt so lange, bis der Endknoten des Bereichs erreicht wurde.
Sind alle drei Zeichen aus dem Drei-Zeichen-Index auf diese Art und Weise verarbeitet worden, so sind alle TSF-Features der Texte der Trainingsdaten der Klasse ermittelt worden und es kann mit den Texten der Trainingsdaten der nächsten Klasse fortgefahren werden, die genauso verarbeitet wird.

1 Zur Erinnerung: Gleiche Suffixe aus zwei verschiedenen Texten wurden bereits beim Erzeugen des GSA, siehe S. 210, als Feature erkannt.

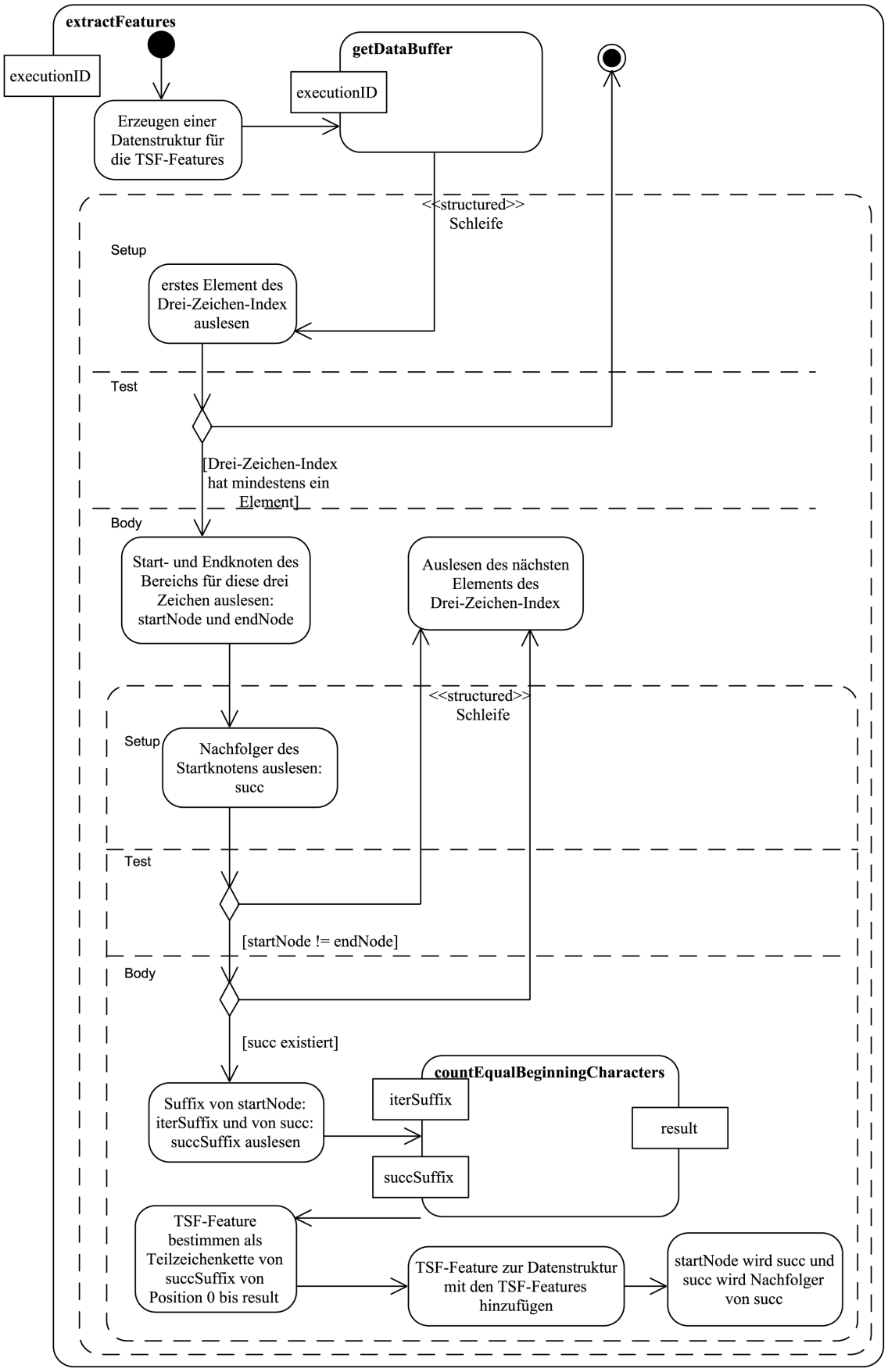


Abbildung 3.36: Ablauf der Operation `extractFeatures`

3.2.3.2 Implementierung der Ermittlung von Text-Suffix-Fragment-Features zum Clustern eines Textes („AllDoks“)

Der erste Ansatz, um die TSF-Features eines Textes für das Clustern dieses Textes zu ermitteln, verwendet den in Kapitel 3.2.2.2 beschriebenen Algorithmus. Das bedeutet, in der Implementierung werden im Prinzip die gleichen Operationen ausgeführt, wie bei der Ermittlung von TSF-Features für das Klassifizieren eines Textes.
Als Erstes muss jedoch aus den vorliegenden einzelnen Suffix Arrays ein generalisiertes Suffix Array erstellt werden. Dazu wird zunächst die Operation `extractFeaturesOfAllDocs` aufgerufen, die in Abbildung 3.37 zu sehen ist. Die Operation setzt voraus, dass die einzelnen Texte bereits als Suffix Arrays aufbereitet wurden. Die Operation liest zunächst den `dataBuffer` aus. Danach ermittelt sie alle vorhandenen Suffix Arrays aus dem Verzeichnis, in dem sie gespeichert sind, und schreibt diese in ein Array, um später darauf zugreifen zu können. Sind alle Suffix Arrays auf diese Weise erfasst worden, wird ein generalisiertes Suffix Array erzeugt, um die einzelnen Suffix Arrays darin verschmelzen zu können.
In einer Schleife wird jedes Element des Arrays, das die einzelnen Suffix Arrays speichert, ausgelesen und das einzelne Suffix Array dem generalisierten Suffix Array hinzugefügt.

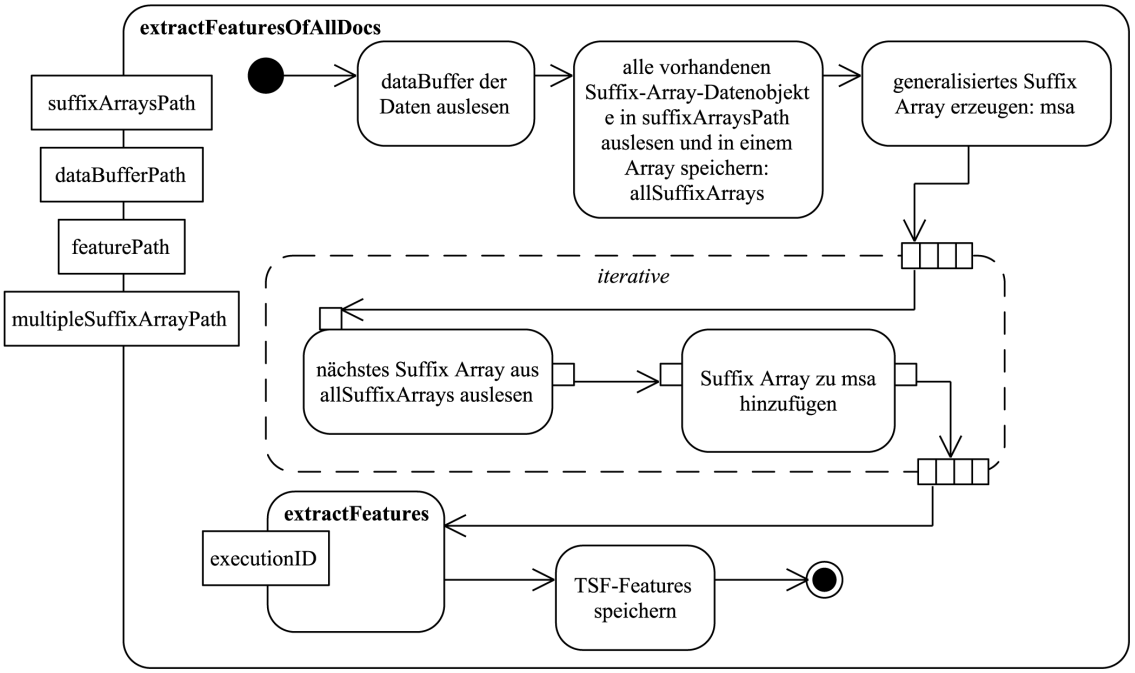


Abbildung 3.37: Ablauf der Operation `extractFeaturesOfAllDocs`

Das erfolgt mit den in Kapitel 3.1.5.3 beschriebenen Operationen. Sind alle einzelnen Suffix Arrays verarbeitet worden, existiert also ein generalisiertes Suffix Array für alle zu clusternden Texte, so kann die Operation `extractFeatures`, die bereits im Kapitel 3.2.3.1 beschrieben wurde, für dieses generalisierte Suffix Array ausgeführt werden. Sie ermittelt alle TSF-Features aller zu clusternden Texte auf die beschriebene Art und Weise und speichert diese anschließend.

3.2.3.3 Implementierung der Ermittlung von Text-Suffix-Fragment-Features zum Clustern eines Textes („SingleDoks“)

Die andere Methode, um die TSF-Features eines Textes für das Clustern desselben zu ermitteln, besteht darin, für jeden der Texte einzeln die TSF-Features zu bestimmen. Im Gegensatz zum Algorithmus wird in der Implementierung jedoch eine effizienzsteigernde Modifikation angewendet, um die Operation zur Ermittlung der Features eines GSA wiederverwenden zu können. Dafür wird das Suffix Array des einzelnen Textes in ein GSA umgewandelt und nicht direkt verarbeitet. Die Operation, die die TSF-Features der einzelnen Texte bestimmt, heißt `extractFeaturesOfSingleDocs`. Sie ist in Abbildung 3.38 auf S. 227 zu sehen. Auch bei dieser Operation ist Voraussetzung, dass die einzelnen Texte bereits als Suffix Array aufbereitet wurden. Die Operation liest zunächst den `dataBuffer` aus. Danach ermittelt sie alle vorhandenen Suffix Arrays aus dem Verzeichnis, in dem sie gespeichert sind, und schreibt diese in ein Array, um später darauf zugreifen zu können. Sind alle Suffix Arrays auf diese Weise erfasst worden, wird jedes Element des Arrays, also jedes Suffix Array für einen einzelnen Text, innerhalb einer Schleife ausgelesen.
Die TSF-Features eines einzelnen Textes lassen sich auf die gleiche Weise ermitteln, wie in Kapitel 3.2.3.1 beschrieben. Dafür benötigt man jedoch ein GSA. Also wird jedes Suffix Array in ein GSA umgewandelt, siehe S. 199, und dann die Operation `extractFeatures`, die auf S. 222 beschrieben ist, ausgeführt. Die so ermittelten TSF-Features werden gespeichert und es wird mit dem nächsten Text fortgefahren.

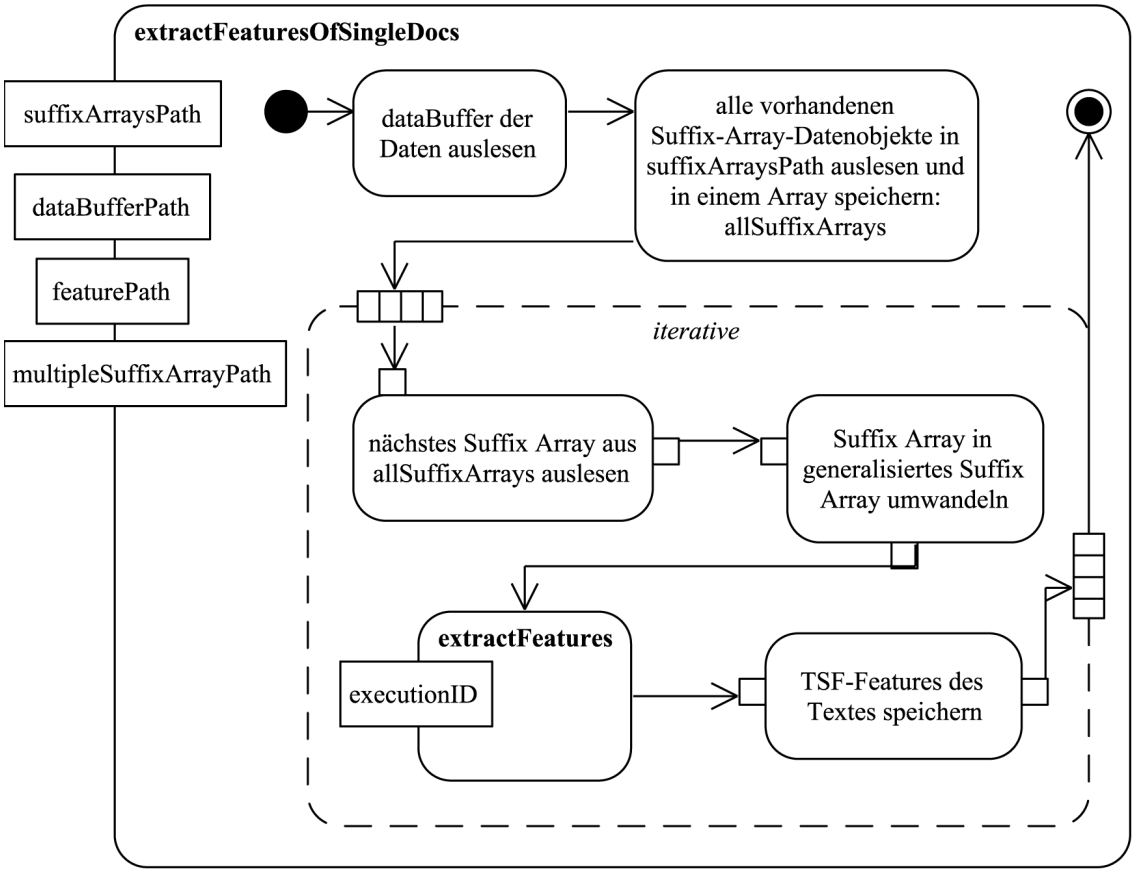


Abbildung 3.38: Ablauf der Operation `extractFeaturesOfSingleDocs`

4 Experimente mit dem Klassifizieren von natürlichsprachlichen Dokumenten

4.1 Implementierung des Klassifizierens mit Text-Suffix-Fragment-Features

4.1.1 Einleitung in die Implementierung des Klassifizierens mit Text-Suffix-Fragment-Features

Ein Ziel der vorliegenden Arbeit ist es zu überprüfen, ob Klassifizierungen von natürlichsprachlichen Dokumenten, basierend auf TSF-Features, bessere Ergebnisse liefern als Klassifizierungen der gleichen Dokumente, basierend auf Worten als Features. Nachdem im vorangegangenen Kapitel beschrieben wurde, wie die TSF-Features für natürlichsprachliche Dokumente ermittelt werden, folgt in diesem Kapitel die Beschreibung des Ablaufs, wie mit diesen Features Dokumente klassifiziert werden. Dabei wird zunächst der zum Klassifizieren verwendete Datensatz vorgestellt und anschließend in jedem Unterkapitel ein Schritt für den Ablauf innerhalb der Implementierung[1] für das Verfahren mit TSF-Features erläutert. Die Beschreibung des Ablaufs für das wortbasierte Verfahren erfolgt an späterer Stelle, siehe Kapitel 4.2.

4.1.2 Beschreibung des beim Klassifizieren verwendeten Datensatzes

4.1.2.1 Vorstellung des Datensatzes

Der in dieser Arbeit verwendete Datensatz ist der Reuters-RCV1-v2-Datensatz.[2] Dieser ist in der dort beschriebenen Form nicht verfügbar, sondern nur eine Vorgängerversion - der RCV1-v1, die vom National Institute of Standards and Technology (NIST)[3] zur Verfügung gestellt wird. Um vergleichbare Ergebnisse zu Lewis

1 Eine Erläuterung zum Software-Prototyp befindet sich in Anhang F auf S. 593.

2 Vgl. Lewis u.a. (2004). In der genannten Quelle befindet sich eine Beschreibung des Datensatzes und die Beschreibung der Umwandlungsschritte von RCV1-v1 in RCV1-v2, die in der vorliegenden Arbeit ebenfalls durchgeführt wurden. Der Datensatz selbst ist weder in der genannten Quelle, noch in online appendizes, noch in der vorliegenden Arbeit enthalten. Das liegt an der rechtlichen Situation, die in NIST (2004) erläutert wird. Der Datensatz muss bei NIST angefordert werden. Nachdem eine Vereinbarung unterschrieben wurde, diesen nur zu wissenschaftlichen Zwecken zu verwenden, erhält man ihn auf mehreren Datenträgern. Die Daten selbst dürfen jedoch nicht abgedruckt werden.

3 Vgl. NIST (2004).

u.a. (2004) zu erhalten, muss diese Vorgängerversion entsprechend den vorgegebenen Schritten[1] angepasst werden. Das wurde im Rahmen der vorliegenden Arbeit wie dort beschrieben durchgeführt, so dass die Ergebnisse auf den gleichen Daten beruhen.
Das Sammeln der Daten und die Klassifizierung wurde von der Nachrichtenagentur Thomson Reuters durchgeführt. Die in Lewis u.a. (2004) beschriebene Dokumentation über die Erzeugung der Testdatensammlung durch Reuters ist anhand von Interviews und Datenanalysen rekonstruiert worden, da Reuters selbst den Prozess nicht dokumentiert hat.
Im Folgenden wird lediglich auf die Version RCV1-v2 Bezug genommen und die Vorgängerversion nicht betrachtet, da sie in dieser Arbeit keine Rolle spielt.

4.1.2.2 Aufbau der Daten

Die Sammlung an sich besteht aus 804.414 Dokumenten mit Nachrichten - im weitesten Sinne - in englischer Sprache, die zwischen dem 20.08.1996 und dem 19.08.1997 erfasst wurden. Ihre Länge variiert zwischen einigen hundert und mehreren tausend Wörtern pro Dokument. Es existiert eine Einteilung[2] in eine Trainings- und eine Testmenge unter chronologischen Gesichtspunkten, das heißt, die Dokumente bis zum 31.08.1996 gehören zur Trainingsmenge (23.149) und die restlichen Dokumente zur Testmenge (781.265).
Es existieren drei Klassenfamilien[3]:

- Topic,
- Industry und
- Region.

Das ist in Abbildung 4.1 auf S. 231 zu sehen.
Mit Hilfe der Topic-Klassen wird der Hauptinhalt des Dokuments beschrieben, d.h., jedes Dokument soll mindestens einem Topic zugeordnet sein[4]. Die Topic-Klassen sind hierarchisch über 4 Ebenen aufgebaut. Sie unterteilen sich je nach Ebene in unterschiedlich viele Klassen. Dabei ist diese Aufteilung weder einheitlich strukturiert, noch folgt sie einheitlichen Bezeichnungen. Das kann man der Abbildung 4.2 auf

1 Vgl. Lewis u.a. (2004), S. 379 f.
2 Diese Einteilung wurde im Zuge der Erzeugung der Version RCV1-v2 vorgenommen, vgl. Lewis u.a. (2004), S. 380, und auch in dieser Arbeit verwendet.
3 In den folgenden Kapiteln werden die Klassenfamilien auch als *Familien von Klassen* bezeichnet. Die Erläuterungen der Bedeutung der Klassenbezeichnungen befindet sich in den folgenden Abschnitten.
4 Das wird „Minimum Code Policy“ genannt, Lewis u.a. (2004), S. 366.

S. 231 entnehmen. Hierbei ist zu beachten, dass die Hierarchie hier nicht vollständig wiedergegeben wird. Die Auslassungspunkte verdeutlichen, an welchen Stellen weitere Hierarchiebenen von der Verfasserin ausgelassen wurden.[1]

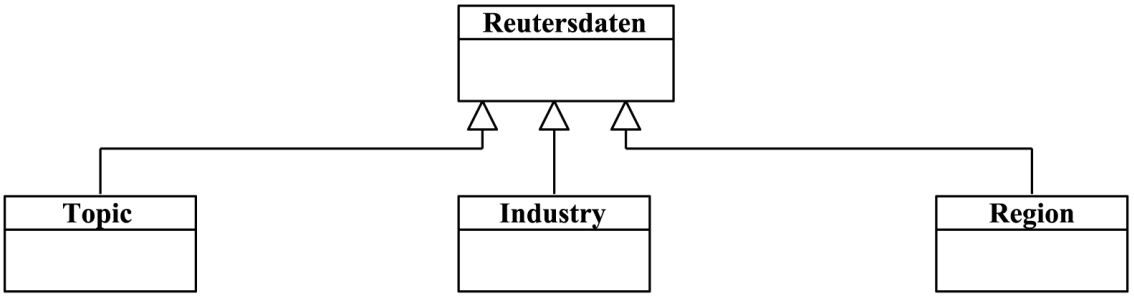


Abbildung 4.1: Aufteilung der Reuters-Daten in drei Klassenfamilien

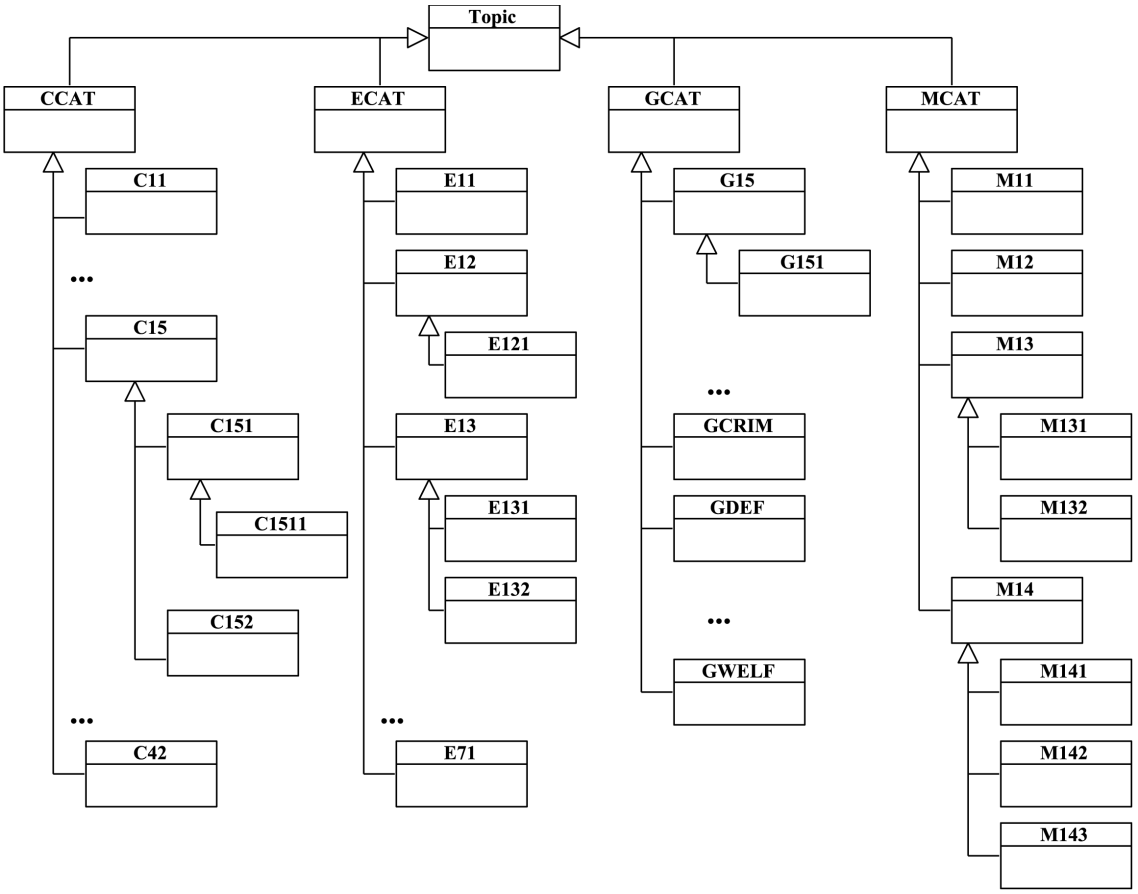


Abbildung 4.2: Hierarchie der Klassenfamilie Topics der Reuters-Daten

1 Bei den vermeintlichen Akronymen der Klassennamen in der Abbildung handelt es sich nicht um Akronyme, sondern um die von Reuters vergebenen Klassennamen.

Die Industry-Klassen beschreiben die Branchen, auf die im Dokument Bezug genommen wird. Hier existieren 10 Unterklassen, die teilweise ebenfalls hierarchisch aufgebaut sind[1]. Einem Dokument muss keine Industry-Klasse zugeordnet werden. Jedes Dokument soll mindestens einer der 366 Region-Klassen zugeordnet sein[2]. Zur Entscheidung werden die im Dokument vorhandenen geografischen Lageinformationen verwendet. Bei dieser Klassenfamilie sind keine weiteren Hierarchiestufen vorhanden.
Des Weiteren gilt als Vorgabe für die Zuordnung zu Klassen, dass die Dokumente den speziellsten Klassen zugeordnet werden und deren Oberklassen ebenfalls alle hinzugefügt werden.

4.1.3 Erzeugen von Reuters Corpus Volume 1 Version 2

4.1.3.1 Einleitung in die Erzeugung von Reuters Corpus Volume 1 Version 2

In der vorliegenden Arbeit wird in den Experimenten der Reuters-RCV1-v2-Datensatz verwendet.[3] Der vom NIST zur Verfügung gestellt Reuters-Datensatz[4] enthält noch einige Unstimmigkeiten. Diese wurden von Lewis u.a. (2004) festgestellt und vor den Tests mit dem Datensatz beseitigt, so dass eine verbesserte Version des Datensatzes - der RCV1-v2 - entstand.[5] Diese Unstimmigkeiten müssen auch hier beseitigt werden, um die Ergebnisse[6] vergleichbar zu halten. Die dafür nötigen Veränderungen werden in den folgenden Abschnitten beschrieben.

4.1.3.2 Entfernen von Dokumenten ohne Region-Codes

Laut Lewis u.a. (2004)[7] befinden sich unter allen Dokumenten 13, die keinen Region-Code erhalten haben. Da jedoch jedem Dokument mindestens ein Region-Code zugeordnet werden muss, werden diese Dokumente aus dem Datensatz entfernt.

1 Aufgrund der vorhandenen Anomalien in Bezug auf diese Klassenfamilie verwenden Lewis u.a. (2004) nicht alle dieser Klassen, vgl. Lewis u.a. (2004), S. 371, 373 f. Die weitere Hierarchisierung der Klassenfamilie ist ebenfalls uneinheitlich. Sie ist aber der Topic-Hierarchisierung sehr ähnlich.
2 Ebenfalls der „Minimum Code Policy“ folgend, Lewis u.a. (2004), S. 366.
3 Eine Beschreibung des Datensatzes befindet sich in Kapitel 4.1.2.
4 Vgl. NIST (2004).
5 Vgl. Lewis u.a. (2004), S. 379.
6 In der vorliegenden Arbeit werden in den durchgeführten Experimenten die von Lewis (2004) zur Verfügung gestellten Feature-Vektoren verwendet. Diese beruhen auf dem bereinigten Datensatz RCV1-v2. Damit die zu erstellenden TSF-Feature-Vektoren ebenfalls auf dem RCV1-v2-Datensatz beruhen, muss dieser in der vorliegenden Arbeit ebenfalls erstellt werden.
7 Vgl. Lewis u.a. (2004), S. 379.

Da dies in der Java-Implementierung zusammen mit dem Entfernen von Dokumenten ohne Topic-Codes, siehe Kapitel 4.1.3.3, durchgeführt wird, können die dafür nötigen Schritte dort nachgelesen werden.

4.1.3.3 Entfernen von Dokumenten ohne Topic-Codes

Es existieren 2.364 Dokumente im Datensatz, die während des Klassifizierens durch Reuters keine Topic-Codes erhalten haben.[1] Auch hier gilt die Vorschrift, dass jedem Dokument mindestens einer dieser Codes zugeordnet werden muss. Diese Dokumente müssen ebenfalls entfernt werden. Dafür werden in der Operation `createRCV1V2-ByDelete`, zu sehen in Abbildung 4.3, alle Dokumente durchlaufen.

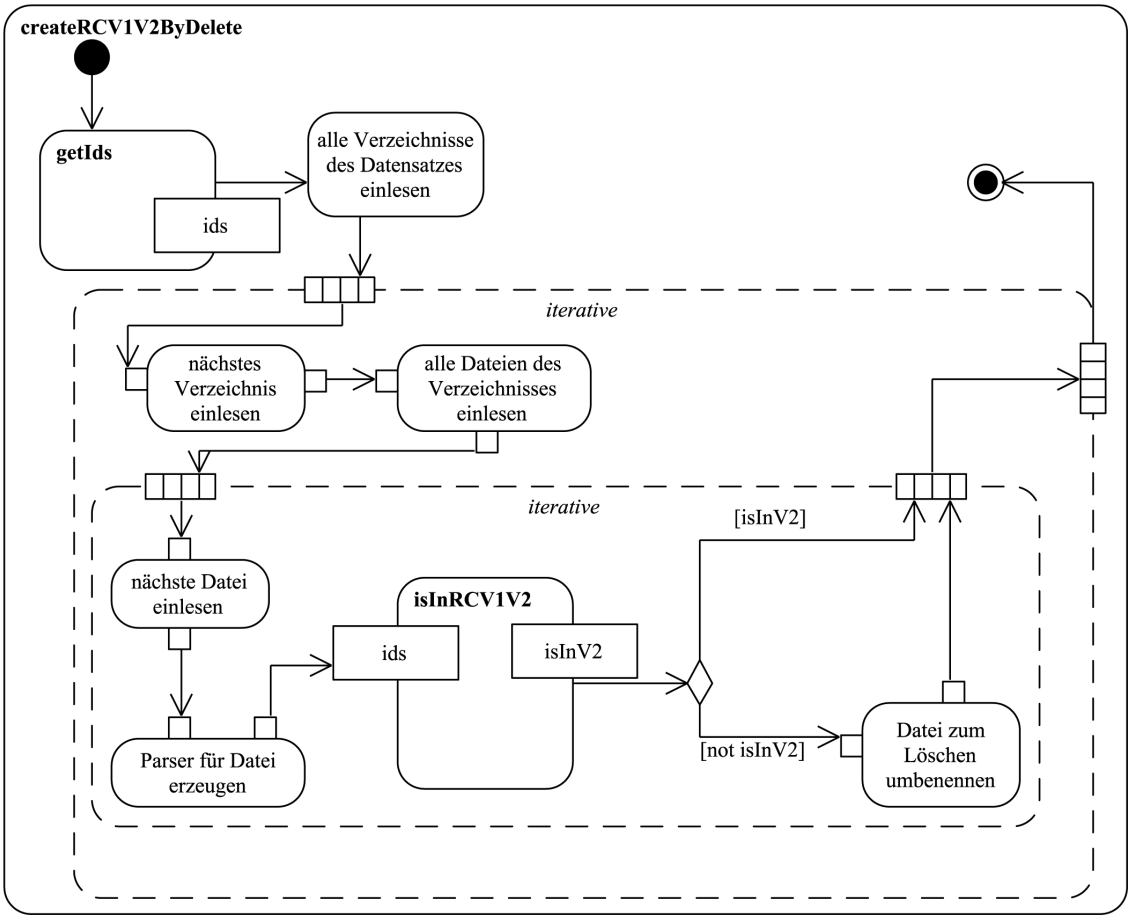


Abbildung 4.3: Ablauf der Operation `createRCV1V2ByDelete`

Jedes einzelne Dokument wird in der Operation `isInRCV1V2`, Abbildung 4.5, S. 234, daraufhin überprüft, ob seine Reuters-ID in den Reuters-IDs des RCV1-v2 enthalten

1 Vgl. Lewis u.a. (2004), S. 379.

ist. Diese werden von D.D.Lewis zur Verfügung gestellt.[1] Es erfolgt also ein Einlesen dieser im RCV1-v2 enthaltenen IDs aus der entsprechenden Datei. Dann wird mit Hilfe eines XML-Parsers[2] für jede ID der Dokumente das Vorhandensein der Dokument-ID in der Liste der RCV1-v2-IDs überprüft. Sollte die ID eines Dokuments nicht vorhanden sein, so wird die Datei dieses Dokuments umbenannt. Nach der Überprüfung aller Dokumente werden schließlich alle umbenannten Dateien aus dem Datensatz entfernt.

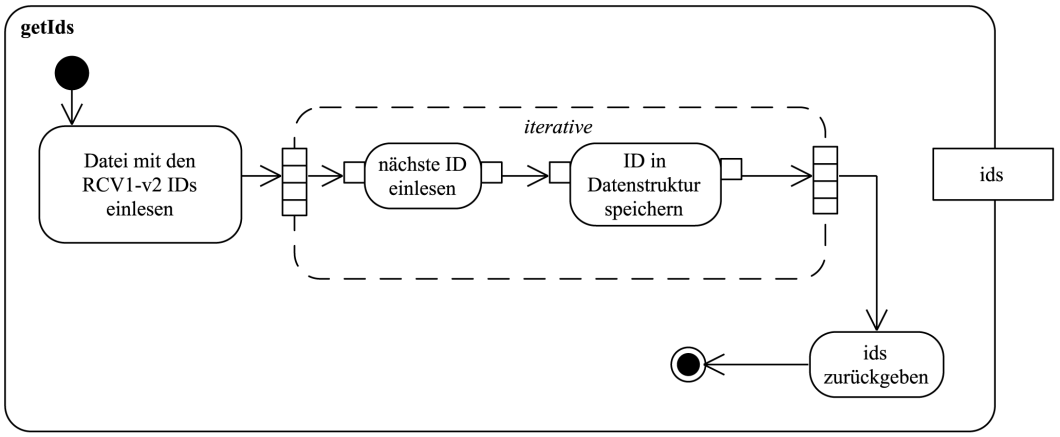


Abbildung 4.4: Ablauf der Operation `getIds`

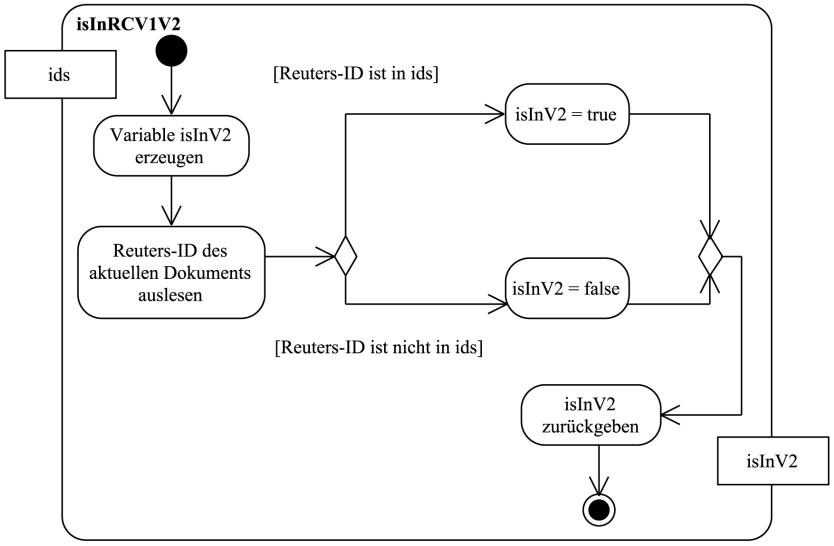


Abbildung 4.5: Ablauf der Operation `isInRCV1V2`

1 Vgl. Lewis (2004), Online-Appendix 7.
2 Eine Erläuterung eines solchen Parsers kann in Kapitel 4.1.5.2 nachgelesen werden.

4.1.3.4 Ergänzen aller Vorfahren bei Topic-Codes

Das Klassifizieren der Reuters-Daten soll die Hierarchie der Topic-Codes berücksichtigen.

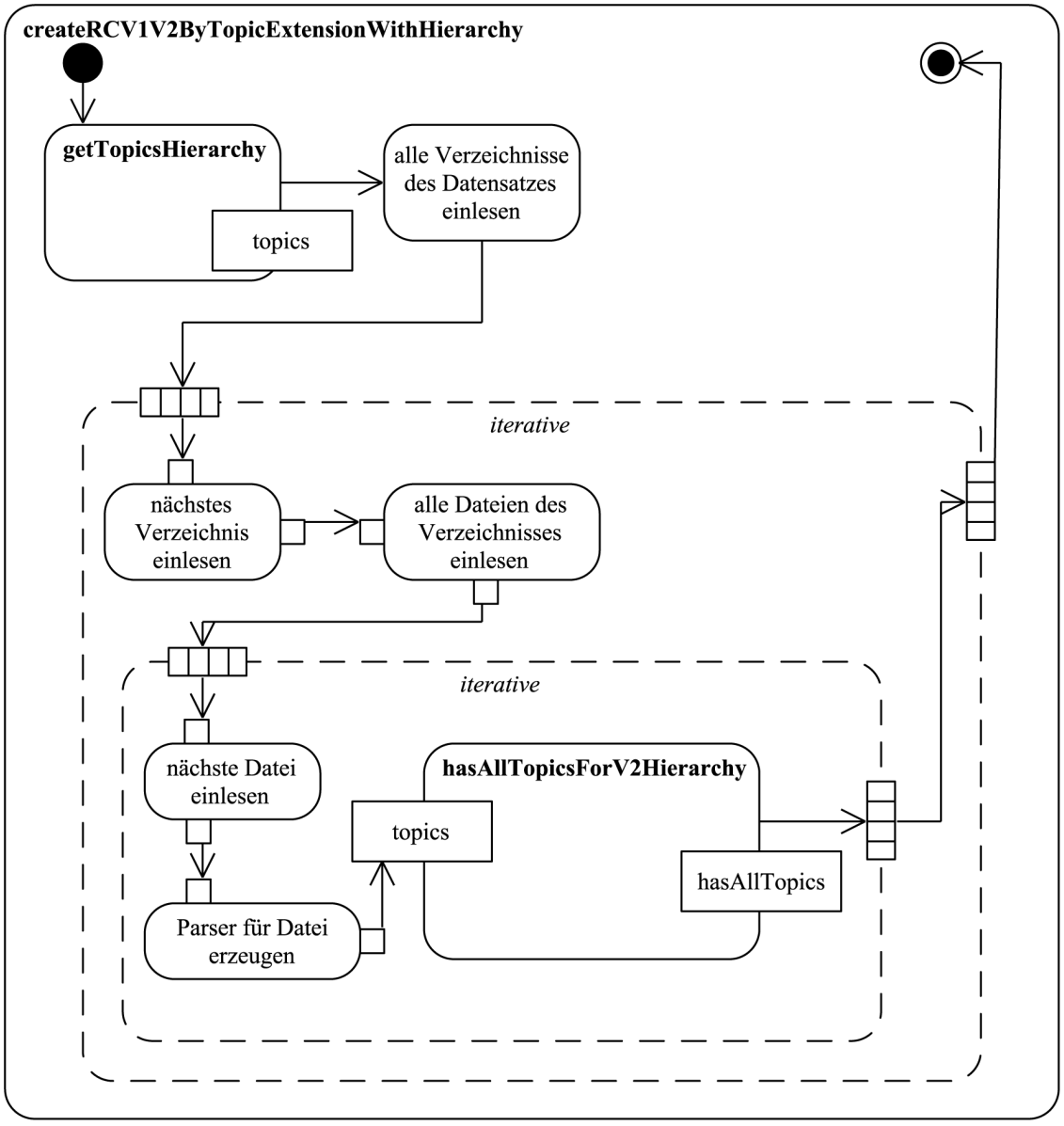


Abbildung 4.6: Ablauf der Operation `createRCV1V2ByTopicExtensionWithHierarchy`

Damit die Berücksichtigung gewährleistet ist, müssen bei allen Dokumenten alle Codes der übergeordneten Ebenen vorhanden sein. Das ist jedoch im RCV1-v1 nicht der Fall. Aus diesem Grund ist es nötig, die fehlenden übergeordneten Topic-Codes in den Dokumenten zu ergänzen, in denen sie fehlen.

In der Operation `createRCV1V2ByTopicExtensionWithHierarchy`, zu sehen in Abbildung 4.6 auf S. 235, wird deshalb zuerst die Operation `getTopicsHierarchy` aufgerufen. Dort wird, wie in Abbildung 4.7 zu sehen, die vorgegebene Hierarchie der Topic-Codes[1] eingelesen. Die Topic-Codes werden in einer Datenstruktur gespeichert, die auf Schlüssel-Wert-Paaren basiert. Der jeweilige Kind-Code ist der Schlüssel und der nächste übergeordnete Vater-Code der Wert. Bei den vier Codes der obersten Ebene - CCAT, ECAT, GCAT und MCAT - wird als Vater-Code „root“ gespeichert. Danach werden alle Dokumente durchlaufen und mit der Operation `hasAllTopicsForV2Hierarchy`, siehe Abbildung 4.8 auf S. 237, daraufhin überprüft, ob sie alle Hierarchiebenenen ihrer Topic-Codes enthalten. Sollte das nicht der Fall sein, werden die fehlenden Codes der XML-Datei des Dokuments hinzugefügt.

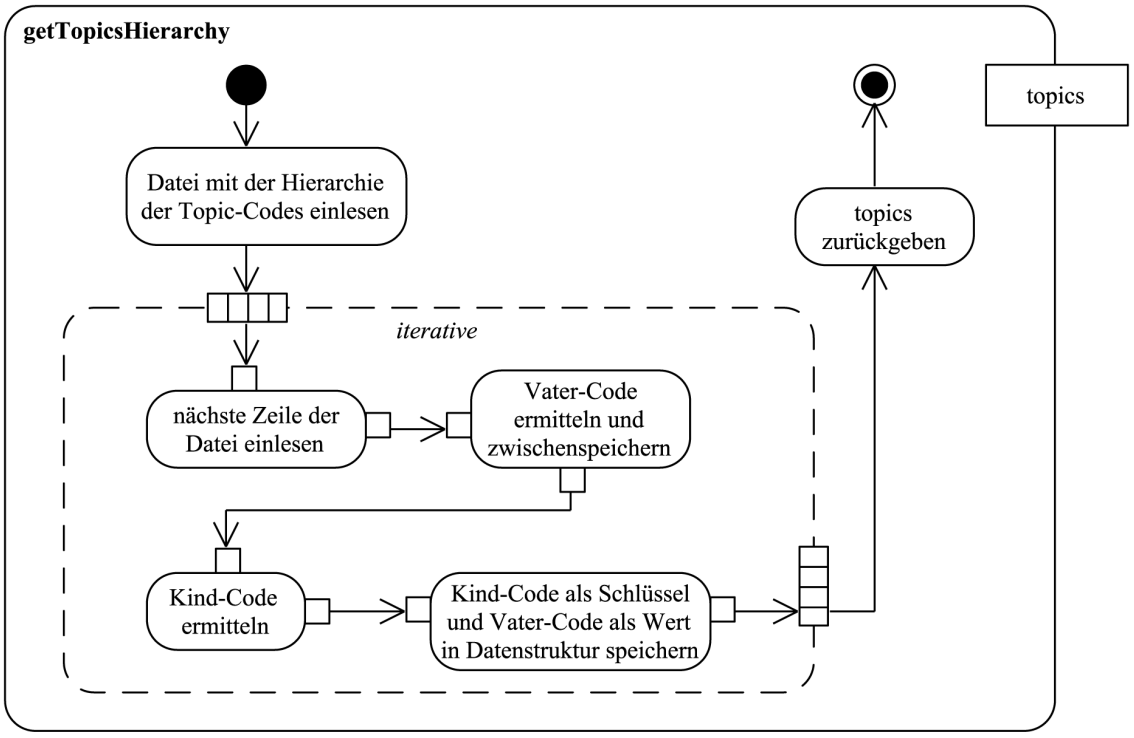


Abbildung 4.7: Ablauf der Operation `getTopicsHierarchy`

1 Vgl. Lewis (2004), Online-Appendix 2.

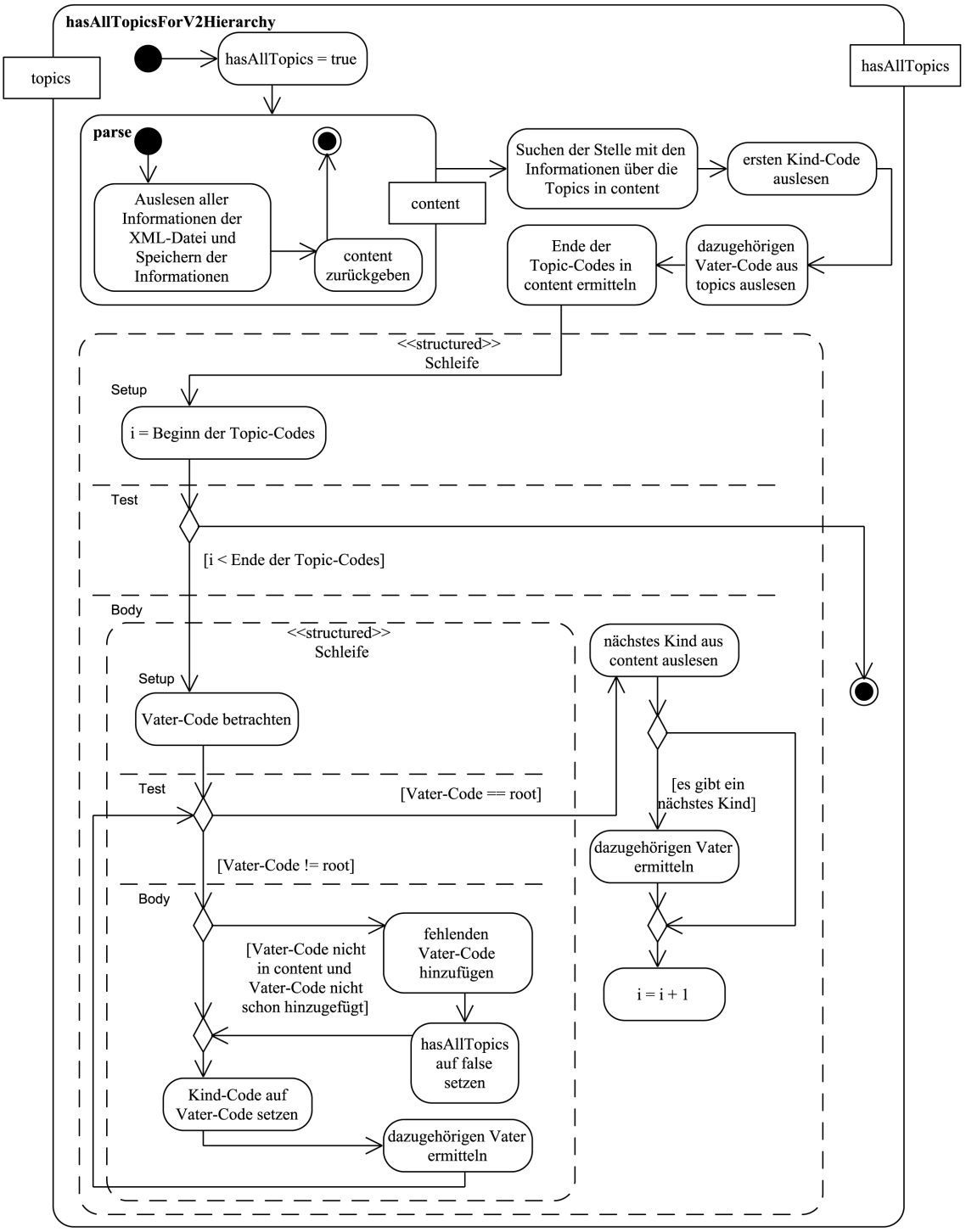


Abbildung 4.8: Ablauf der Operation `hasAllTopicsForV2Hierarchy`

4.1.3.5 Verbessern von falschen Region-Codes

In den Reuters-Daten sind falsche Region-Codes vorhanden.[1] Das bedeutet, 4 Dokumente wurden Region-Klassen zugeordnet, die es nicht gibt. Diese falschen Codes müssen durch die richtigen ersetzt werden, um den korrekten RCV1-v2-Datensatz zu erhalten.
Aus diesem Grund werden alle XML-Dateien innerhalb der Operation `createRCV1V2BySearchingFalseRegionCodes`, zu sehen in Abbildung 4.9 auf S. 239, durchlaufen, geparst und daraufhin überprüft, ob sie einen der falschen Codes: *CZ, CZECH* oder *GDR* enthalten. Die Überprüfung erfolgt in der Operation `hasFalseRegionCode`, siehe Abbildung 4.10 auf S. 240.
Wird ein solcher falscher Region-Code gefunden, wird die entsprechende XML-Datei gekennzeichnet. Nach dem Durchlaufen aller XML-Dateien wird in den 4 gefundenen Dateien per Hand der falsche Region-Code durch den richtigen ersetzt.[2] Damit ist das Erzeugen des RCV1-v2-Datensatzess aus dem RCV1-v1-Datensatz abgeschlossen.

1 Vgl. Lewis u.a. (2004), S. 379.
2 Bei nur 4 Dateien ist eine Verbesserung der falschen Region-Codes per Hand zu vertreten, da es sich nur um eine kleine, einmal durchzuführende Änderung handelt. Hierbei wird *CZ* durch *PANA*, *CZECH* durch *CZREP* und *GDR* durch *GFR* ersetzt, vgl. Lewis u.a. (2004), S. 374.

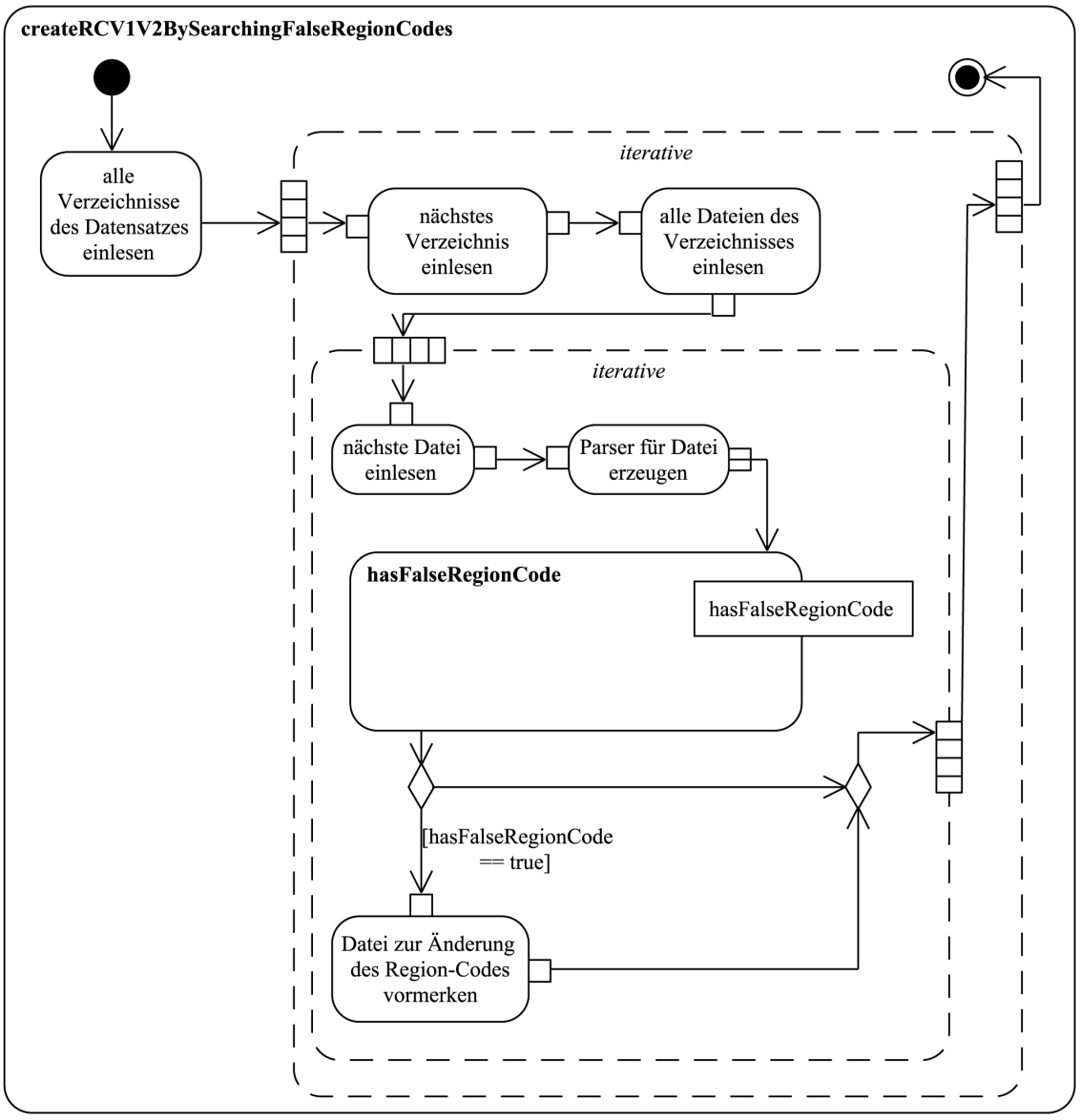


Abbildung 4.9: Ablauf der Operation `createRCV1V2BySearchingFalseRegionCodes`

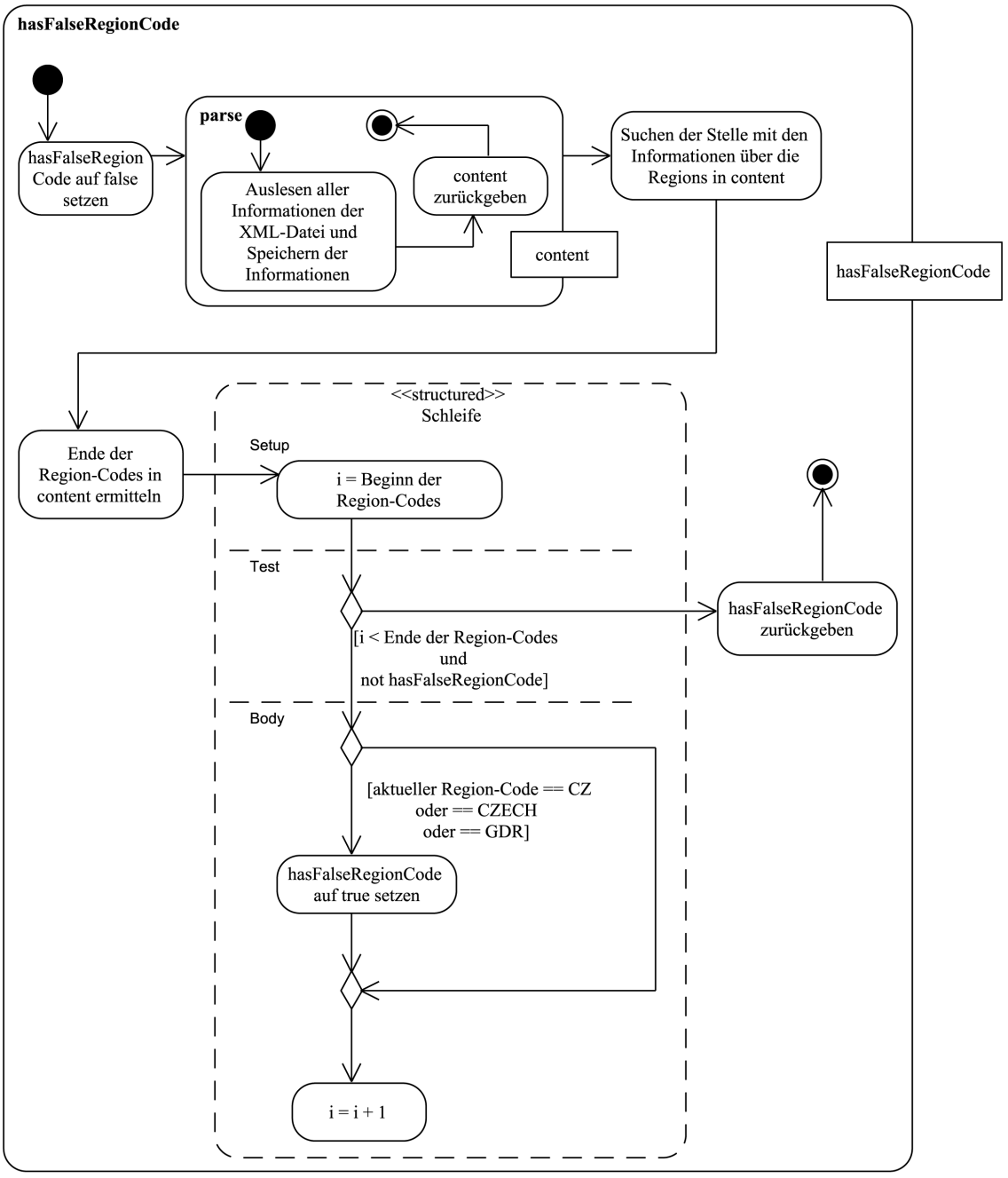


Abbildung 4.10: Ablauf der Operation `hasFalseRegionCode`

4.1.4 Aufteilen des Datensatzes in Trainings- und Testdaten

Um eine durchgeführte Klassifizierung auf ihre Güte hin überprüfen zu können, muss eine Trennung zwischen den Daten, mit denen der Klassifizierer trainiert wurde, und den Daten, die klassifiziert werden, vorliegen. Die hier vorgenommene Teilung des Reuters-Datensatzes RCV1-v2 entspricht der von Lewis u.a. (2004) beschriebenen und verwendeten.[1]

Dafür werden alle XML-Dateien in der Operation `createRCV1V2BySplittingTrainingTest`, zu sehen in Abbildung 4.11, durchlaufen und anhand ihrer Reuters-ID überprüft, ob sie zur Menge der Trainingsdaten oder zur Menge der Testdaten gehören. Diese Überprüfung geschieht innerhalb der Operation `copyFile`, siehe Abbildung 4.13 auf S. 242. Wie in Lewis u.a. (2004), S. 380, beschrieben, reichen die Reuters-IDs der Trainingsdaten von *2.286* bis *26.150* und die der Testdaten von *26.151* bis *810.596*. Pro XML-Datei wird also die Reuters-ID des entsprechenden Dokuments ermittelt und die Datei selbst in ein entsprechend bezeichnetes - „Training“ oder „Test“ - Verzeichnis kopiert. Damit ist die Aufteilung in Trainings- und Testdaten des RCV1-v2-Datensatzes durchgeführt.

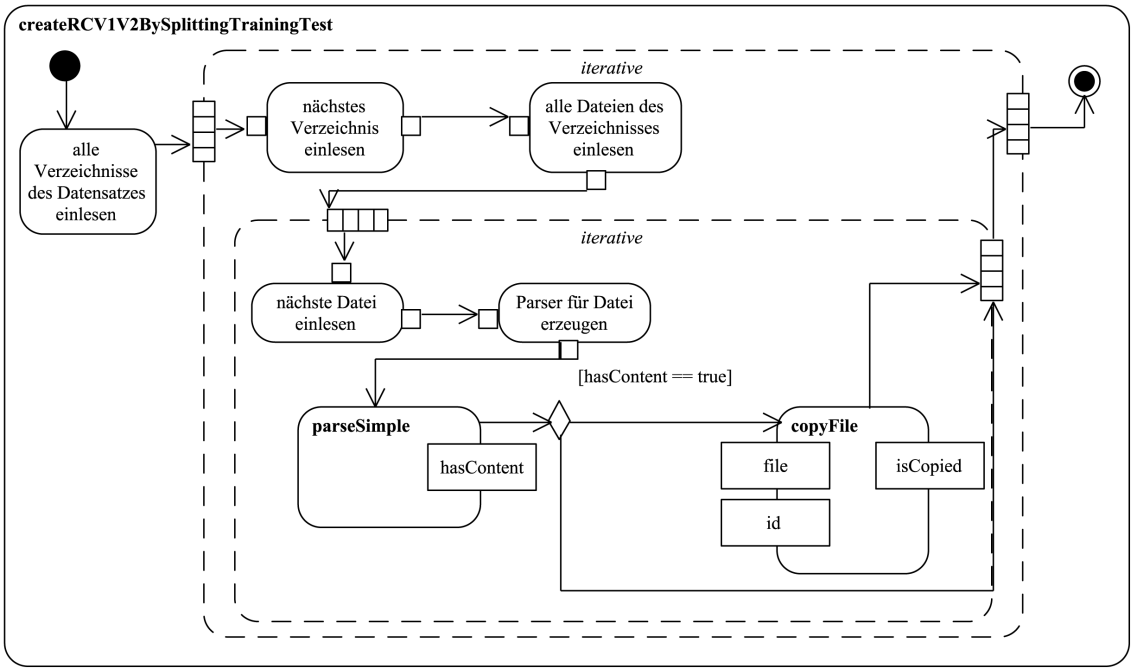


Abbildung 4.11: Ablauf der Operation `createRCV1V2BySplittingTrainingTest`

1 Vgl. Lewis u.a. (2004), S. 380.

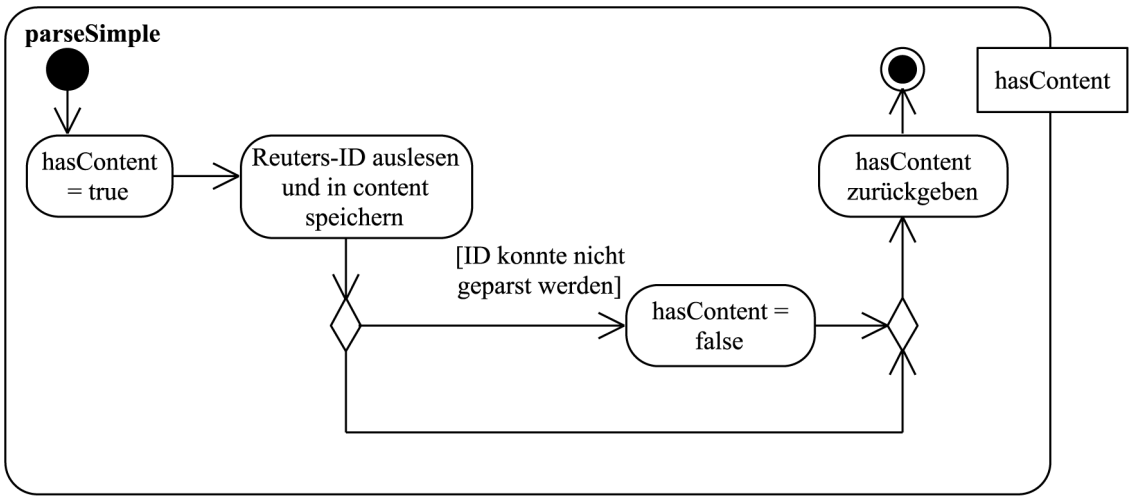


Abbildung 4.12: Ablauf der Operation `parseSimple`

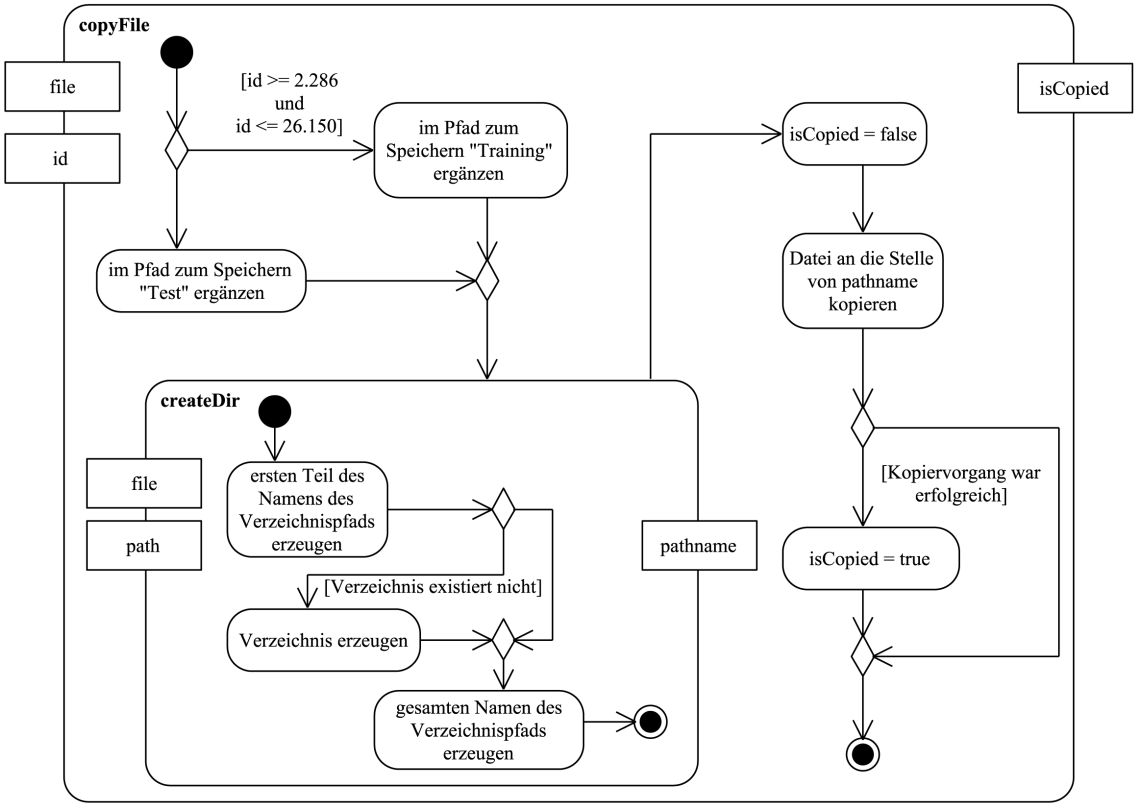


Abbildung 4.13: Ablauf der Operation `copyFile`

4.1.5 Ursprüngliche Reuters-Daten umwandeln

4.1.5.1 Einleitung in die Umwandlung der ursprünglichen Reuters-Daten

Eine Beschreibung des hier verwendeten Reuters-Datensatzes RCV1-v2 befindet sich in Kapitel 4.1.2. Hier wird nun erläutert, wie die eigentlichen Dokumente, die in Form von XML-Dateien[1] vorliegen, in Daten überführt werden, die innerhalb des Klassifizierens verwendet werden können. Der erste Schritt ist das Parsen[2] dieser XML-Dateien und das Extrahieren der benötigten Informationen. Um einen geeigneten Parser zu programmieren, muss zunächst der Aufbau der XML-Dateien analysiert werden.

Jede der Dateien beginnt mit dem typischen Vorspann, der die XML-Version und das Encoding[3] der vorliegenden XML-Datei definiert. Das nächste Tag „newsitem“ beinhaltet das eigentliche Dokument und seine Attribute, die teilweise auch für das Klassifizieren verwendet werden. Das wichtigste Attribut dieses Tags ist die „itemid“. Dabei handelt es sich um eine eindeutige, von Reuters vergebene ID für das Dokument. Das nächste Tag - „title“ - beinhaltet den Titel des Dokuments. Hiermit ist nicht die eigentliche Überschrift des Textes gemeint, sondern ein während des Klassifizierens durch Reuters vergebener Titel. Dieser beinhaltet nicht nur die Überschrift des Dokuments, sondern auch Zusatzinformationen, also Metadaten, wie bspw. die zugehörige Region-Klasse. Die eigentliche Überschrift des Dokuments ist im Tag „headline“ vorhanden. Ergänzt wird sie durch das Tag „dateline“, das neben dem Entstehungsdatum des Textes die Stadt oder das Gebiet enthält, wo der Nachrichtentext entstanden ist.

Nachfolgend befindet sich im Tag „text“ der eigentliche Text des Dokuments. Dieser kann durch „p“-Tags in Abschnitte unterteilt sein. Daran schließt das Tag „copy-

1 XML ist die Abkürzung von *Extensible Markup Language*. Die Auszeichnungssprache wurde als Teilmenge von SGML (Standard Generalized Markup Language) vom World-Wide Web Consortium (W3C) entwickelt, vgl. bspw. World-Wide Web Consortium (2008); Meier (2010), S. 134 f. Bei einer XML-Datei handelt es sich um ein Textdokument, das *wohlgeformt*, wie in der Spezifikation definiert, vgl. World-Wide Web Consortium (2008), ist. Ein solches Textdokument besteht aus einem Vorspann und aus einem oder mehreren Elementen. Diese Elemente beginnen mit einem so genannten Start-Tag und enden mit einem End-Tag. Das Start-Tag ist dabei definiert wie in World-Wide Web Consortium (2008) angegeben. Das End-Tag erhält den gleichen Namen wie das Start-Tag und ein vorangestelltes „/“. Jedes Element kann Attribute erhalten. Zwischen dem Start- und dem End-Tag befindet sich der Inhalt des Elements.

2 Unter *Parsen* eines XML-Dokuments versteht man das Durchlaufen des Dokuments mit Hilfe einer Software, eines so genannten *Parsers*. Während dieses Durchlaufs werden die enthaltenen XML-Elemente mit ihrem Inhalt so aufbereitet, dass sie weiterverarbeitet werden können.

3 Bei *encoding* handelt es sich um ein Attribut des XML-Elements einer jeden XML-Datei. Es dient dazu, den benutzten Zeichensatz anzuzeigen, um bspw. Umlaute korrekt anzeigen lassen zu können, vgl. Koch (2010), S. 28.

right“ an, das enthält, wer das Copyright an diesem Dokument besitzt. Damit sind alle Informationen, die aus dem ursprünglichen Nachrichtentext stammen, erfasst. Zusätzlich befinden sich im Anschluss daran die Metadaten des Dokuments, die während des Klassifizierens durch Reuters diesem Dokument hinzugefügt wurden. Die für die vorliegende Arbeit entscheidenden Metadaten befinden sich in den „codes“-Tags, die „dc“-Tags sind hier nicht relevant. Von den „codes“-Tags können drei vorhanden sein, die jeweils durch ihr „class“-Attribut beschreiben, zu welchen Klassen welcher Familie von Reutersklassen, die nachfolgend aufgelistet sind, das Dokument gehört. Die drei Möglichkeiten sind:

- Topic, angegeben im „class“-Attribut durch „bip:topics:1.0“
- Region, angegeben im „class“-Attribut durch „bip:countries:1.0“ und
- Industry, angegeben im „class“-Attribut durch „bip:industries:1.0“.

Innerhalb dieser „codes“-Tags sind einzelne „code“-Tags vorhanden, die jeweils eine Klassenzuordnung des Dokuments zu einer Klasse der Klassenfamilien, enthalten. Auch hier wird die Klassenzugehörigkeit durch ein Attribut namens „code“ im „code“-Tag spezifiziert.
In Abbildung 4.14 auf S. 245 sieht man anhand einer der XML-Dateien von Reuters den Aufbau im Original.

4.1.5.2 Parsen und Konvertieren der XML-Dateien

Jede der XML-Dateien wird von einem Parser geparst. Dabei handelt es sich um einen Parser, der auf `JDom` basiert.[1] Dieser Parser baut eine Baumstruktur der XML-Datei auf, indem alle Tags mit ihren Inhalten hierarchisch aufgelistet werden. Mit Hilfe dieser Struktur ist es möglich, von der Wurzel ausgehend, jedes gewünschte Tag zu erreichen und den Inhalt einer XML-Datei in Daten umzuwandeln, die während des Klassifizierens verwendet werden können.
Das bedeutet hier, dass die von Reuters vergebene ID des Dokuments, der Inhalt des Tags „headline“, also die Überschrift des Nachrichtentextes, und der eigentliche Text ausgelesen und gespeichert werden. Das geschieht innerhalb der Operation `parseForText`, zu sehen in Abbildung 4.17 auf S. 247. Zusätzlich werden in einem nächsten Schritt die Metadaten verarbeitet, indem alle Klassenfamilien und die

1 Bei `JDom` handelt es sich um eine Java-Klasse, die einen DOM-Parser implementiert. DOM bedeutet *Document Object Model* und ist ein W3C-Standard, vgl. World-Wide Web Consortium (2004).

entsprechenden Klassenzuordnungen des Dokuments in der Operation `parseForID-Category`, siehe Abbildung 4.18 auf S. 248, ausgelesen und gespeichert werden.

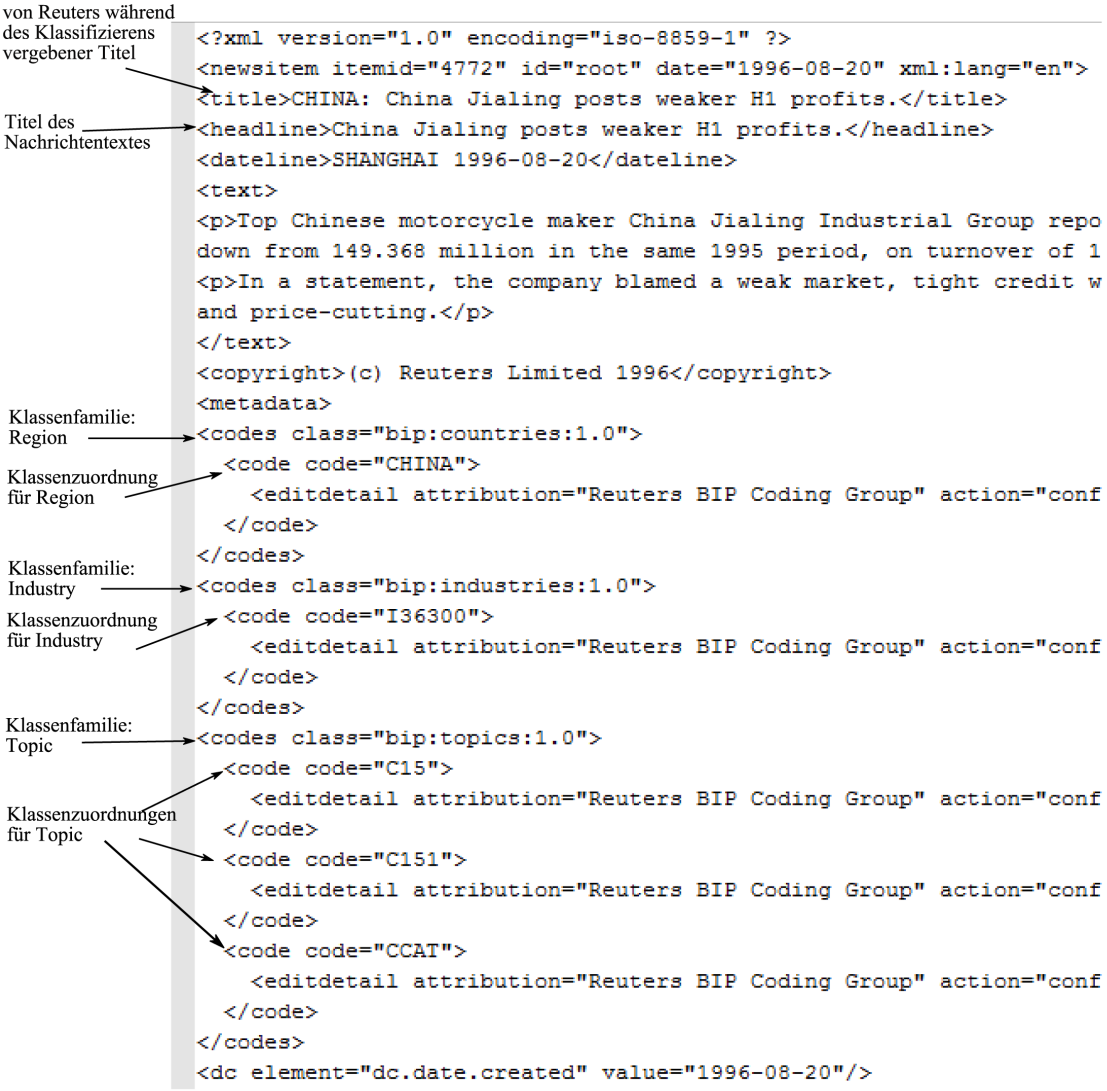


Abbildung 4.14: Ausschnitt einer XML-Datei des Reuters-Datensatzes

Das spätere Klassifizieren soll auf der Überschrift und dem eigentlichen Text des Dokuments basieren. Daher werden diese beiden Teile miteinander verknüpft und konvertiert. Unter der Konvertierung wird in der vorliegenden Arbeit ein Ersetzen aller Leerzeichen, das Entfernen aller Sonder- und Satzzeichen und das Umwandeln aller Groß- in Kleinbuchstaben verstanden. Das Ersetzen der Leerzeichen durch ein spezielles Zeichen ist später für die lexikografische Sortierung wichtig. Das Weglassen der Satz- und Sonderzeichen ebenso wie das Umwandeln aller Groß- in Kleinbuchstaben führt dazu, dass beim Klassifizieren mehr Übereinstimmungen gefunden werden.

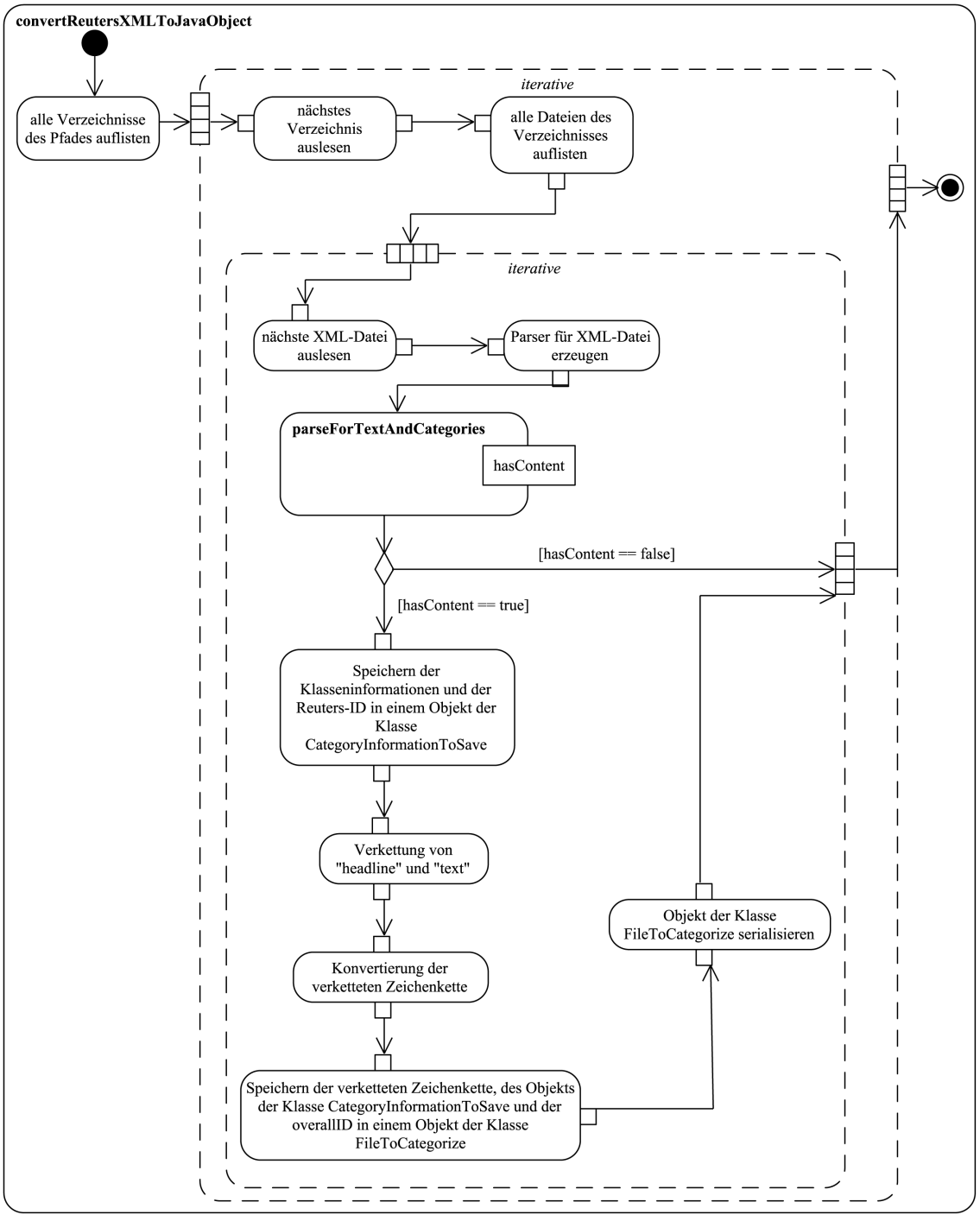


Abbildung 4.15: Ablauf der Operation `convertReutersXMLToJavaObject`

Als letzten Schritt vor der Serialisierung[1] des Java-Objekts mit den aus der XML-Datei extrahierten Daten erhält es eine über alle Dokumente der Reuters-Daten eindeutige und aufsteigende ID.

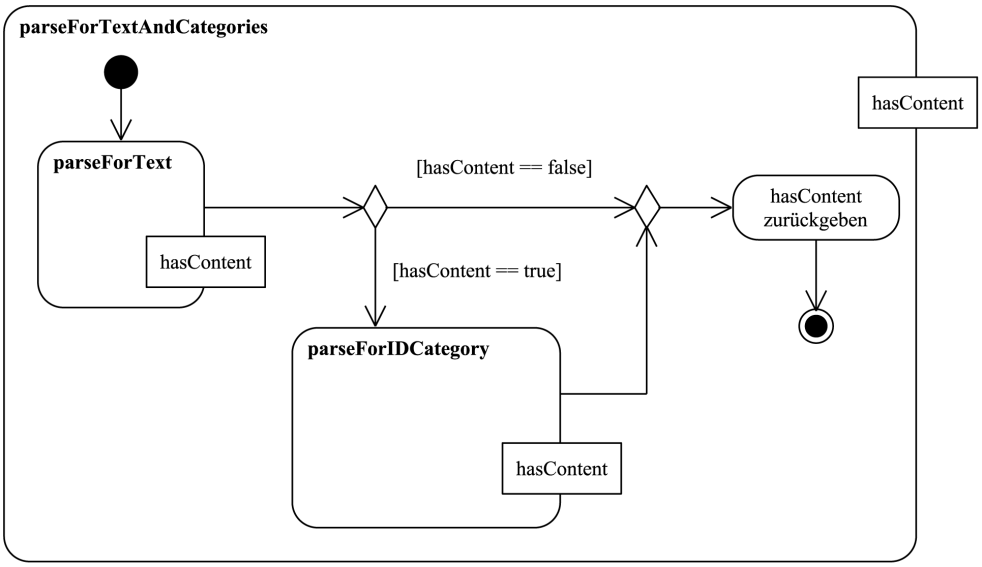


Abbildung 4.16: Ablauf der Operation `parseForTextAndCategories`

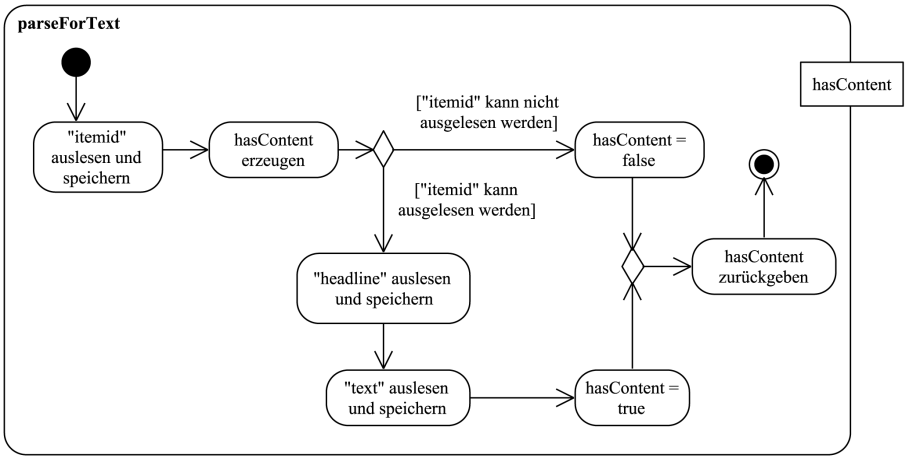


Abbildung 4.17: Ablauf der Operation `parseForText`

1 In der Informatik und insbesondere in der Programmiersprache Java wird der Begriff *Serialisierung* für die Abbildung von Objektstrukturen und ihrer Variablenbelegung auf externe Speicher verwendet. Man sagt auch: die Struktur der Objekte und ihre Variablenbelegung wird *persistiert*, vgl. Ullenboom (2011), S. 909. Die Rekonstruktion zu einem späteren Zeitpunkt wird *Deserialisierung* genannt, vgl. Ullenboom (2011), S. 911.

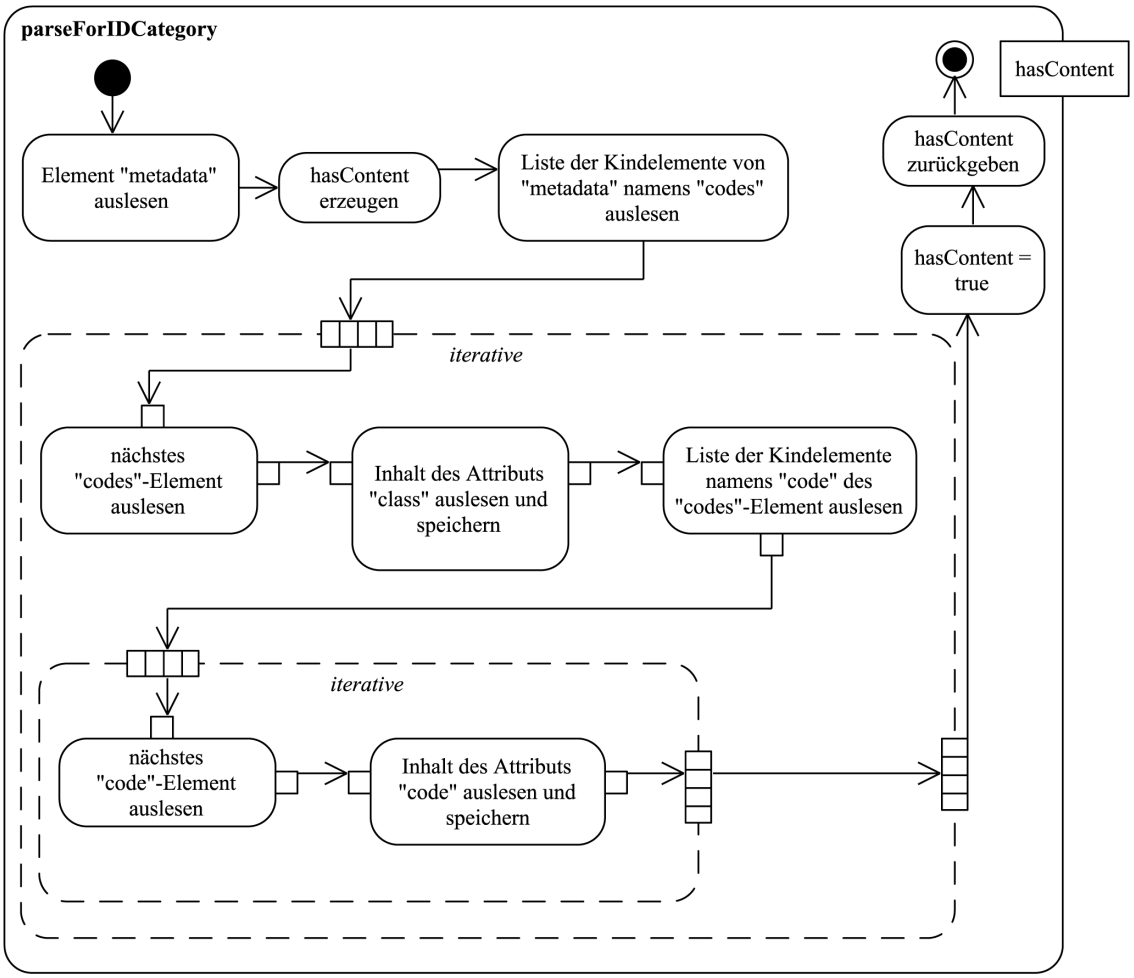


Abbildung 4.18: Ablauf der Operation `parseForIDCategory`

4.1.6 Herausfiltern von Dokumenten mit nur einer Klassenzugehörigkeit

Ein Klassifizieren kann der Zuordnung eines Dokuments entweder zu *genau einer* Klasse oder aber zu *mehreren* Klassen dienen. Um die Güte im Hinblick auf die Zuordnung zu *genau einer* Klasse feststellen zu können, dürfen die zum Trainieren verwendeten Trainingsdaten nur einer Klasse aus einer Klassenfamilie[1] - Region, Topic und Industry - zugeordnet sein.

Um die Dokumente, auf die das Kriterium zutrifft, mehrfach in verschiedenen Teilmengen der Trainingsdaten nutzen zu können, werden alle Dokumente, auf die das Kriterium „nur einer Klasse aus einer Familie von Klassen zugeordnet“ zutrifft, ge-

1 In den Experimenten werden die Klassenfamilien jeweils einzeln betrachtet. Das bedeutet, beim Herausfiltern der Dokumente wird nacheinander deren Klassenzugehörigkeit innerhalb einer der Klassenfamilien geprüft. Bei dieser Überprüfung spielen die Klassenzugehörigkeiten des Dokuments innerhalb der anderen beiden Klassenfamilien keine Rolle.

sucht und entsprechend in einem Verzeichnis gespeichert. Dafür wird innerhalb der Operation `saveDataWithOneClass`, siehe Abbildung 4.19 auf S. 249, die Operation `readFiles`, zu sehen in Abbildung 4.20 auf S. 250, aufgerufen. In dieser Operation werden alle Trainingsdokumente durchlaufen und in der Operation `chooseFile`, zu sehen in Abbildung 4.21 auf S. 250, ihre Klassenzugehörigkeiten ausgelesen. Ist bei einem Dokument für die betrachtete Familie von Klassen nur eine Klassenzuordnung vorhanden, so wird es im entsprechenden Verzeichnis gespeichert.[1]

Auch bei den Testdaten ist es sinnvoll, Teilmengen bilden zu können, deren enthaltene Dokumente nur einer Klasse einer Familie von Klassen zugeordnet sind. Der Ablauf dieses „Filterns“ aufgrund des genannten Kriteriums entspricht dem Ablauf bei den Trainingsdaten. In den Experimenten in der vorliegenden Arbeit werden jedoch auch Testdokumente betrachtet, die in der gerade betrachteten Klassenfamilie mehr als einer Klasse zugeordnet sind, siehe Kapitel 4.3.2 der vorliegenden Arbeit. Daher kann der Operation `chooseFile` der `boolean` Parameter `moreThanOneClass` übergeben werden. Besitzt dieser den Wert *true*, so wird die Datei des Dokuments in jedem Fall gespeichert, da in diesem Fall die Klassenzuordnung keine Rolle spielt. Im Fall der Trainingsdaten besitzt dieser Parameter immer den Wert *false*.

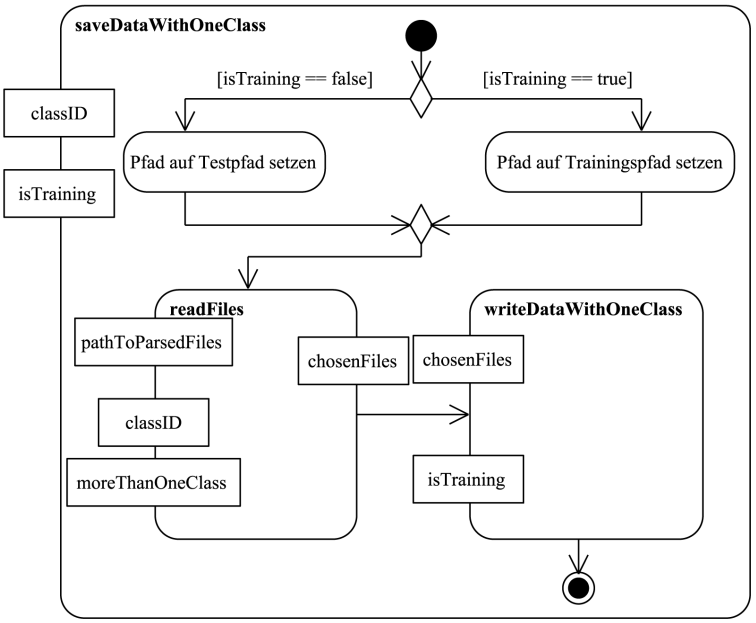


Abbildung 4.19: Ablauf der Operation `saveDataWithOneClass`

1 Die Dokumente, die für die betrachtete Klassenfamilie mehr als eine Klassenzuordnung haben, werden nicht weiter als Trainingsdaten berücksichtigt.

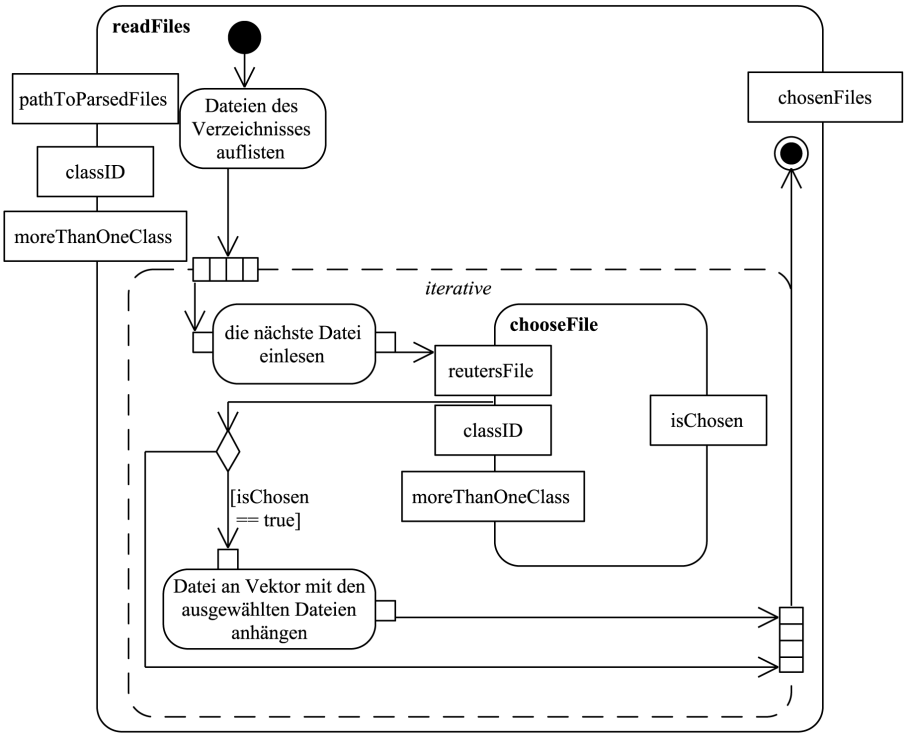


Abbildung 4.20: Ablauf der Operation `readFiles`

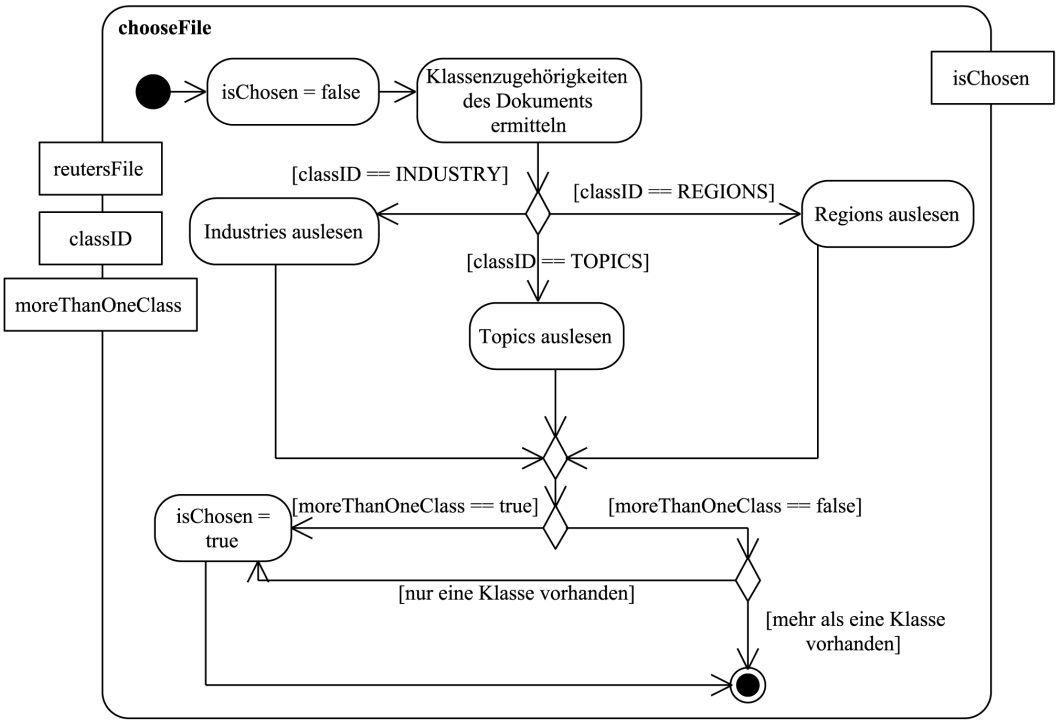


Abbildung 4.21: Ablauf der Operation `chooseFile`

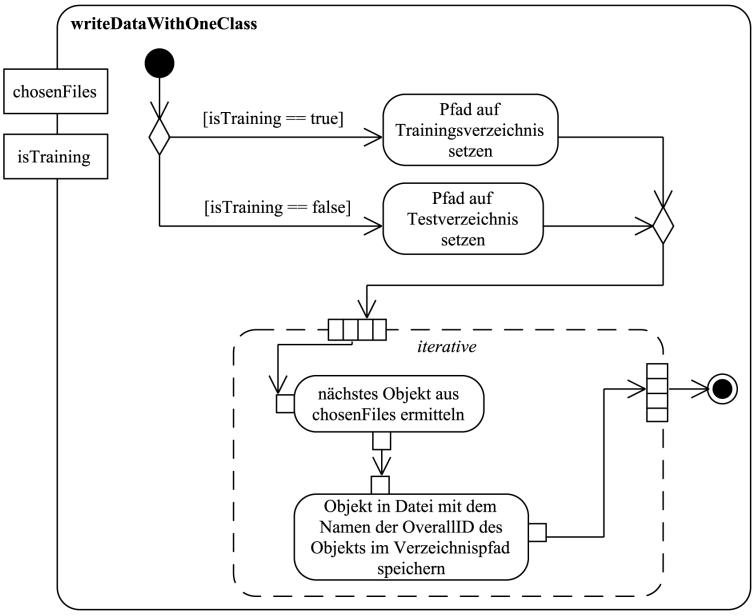


Abbildung 4.22: Ablauf der Operation `writeDataWithOneClass`

4.1.7 Trainingsdaten vorbereiten für das Klassifizieren mit Text-Suffix-Fragment-Features

4.1.7.1 Trainings-Teilmengen erstellen für das Klassifizieren mit Text-Suffix-Fragment-Features

Die im RCV1-v2 vorhandenen 23.149 Trainingsdaten wurden aufgrund ihrer Klassenzugehörigkeit reduziert, da nur Dokumente mit genau einer Klassenzugehörigkeit innerhalb der betrachteten Klassenfamilie im Training benutzt werden. Nachdem alle Dokumente mit nur einer Klassenzugehörigkeit in der betrachteten Klassenfamilie ermittelt worden sind, ergeben sich folgende Anzahlen an Dokumenten:

- Es existieren 15.913 Dokumente, die nur einer Klasse der Klassenfamilie Topic zugeordnet sind.[1]
- Es existieren 7.568 Dokumente, die nur einer Klasse der Klassenfamilie Industry zugeordnet sind.[2]
- Es existieren 18.638 Dokumente, die nur einer Klasse der Klassenfamilie Region zugeordnet sind.

1 Hierbei ist zu beachten, dass sich diese Aussage auf die speziellste Klassenzuordnung des Dokuments bezieht. Die Klassen, die in der Hierarchie über dieser speziellsten Klasse liegen, sind ebenfalls zugeordnet, zählen jedoch nicht als weitere Klassenzuordnungen.

2 Hierbei ist zu beachten, dass sich diese Aussage auf die speziellste Klassenzuordnung des Dokuments bezieht. Die Klassen, die in der Hierarchie über dieser speziellsten Klasse liegen, sind ebenfalls zugeordnet, zählen jedoch nicht als weitere Klassenzuordnungen.

Diese Dokumente werden in Teilmengen aufgeteilt, um die Durchführungsgeschwindigkeit der verschiedenen Experimente zu erhöhen. Die genauen Parameter, also wie viele Teilmengen erstellt werden und wie viele Dokumente diese jeweils umfassen, können den Beschreibungen der jeweiligen Experimente entnommen werden, siehe Kapitel 4.3.2. Die Teilmengen sind durch ein Verzeichnis symbolisiert, in dem sich die ausgewählten konvertierten Reuters-Daten, die zur Teilmenge gehören, befinden. Sie sind als serialisierte Java-Objekte der Java-Klasse `FileToCategorize` vorhanden.

4.1.7.2 Suffix Arrays für die Trainingsdaten für das Klassifizieren mit Text-Suffix-Fragment-Features erstellen

Für jede dieser Teilmengen der Trainingsdaten, die auf die zuvor beschriebene Weise erstellt worden sind, wird für jedes enthaltene und durch das `FileToCategorize`-Objekt repräsentierte Dokument ein Suffix Array erzeugt. Die Erzeugung erfolgt, wie in Kapitel 3.1.4.4 beschrieben. Diese erzeugten Suffix Arrays werden als serialisierte Java-Objekte der Java-Klasse `SuffixArrayToSave` in Bezug auf die Teilmenge, zu der sie gehören, gespeichert.

4.1.7.3 Suffix Arrays nach Klassenzugehörigkeit abspeichern

Um die Erstellung der generalisierten Suffix Arrays zu erleichtern, werden alle Suffix Arrays in der Operation `saveSingleSufArInCategories`, zu sehen in Abbildung 4.23 auf S. 253, eingelesen und aufgrund aller Klassenzugehörigkeiten des durch das Suffix Array repräsentierten Dokuments nochmals abgespeichert. Dabei wird für jede Klasse des RCV1-v2-Datensatzes ein Verzeichnis angelegt. Zu beachten ist hierbei, dass die hierarchische Struktur der Reuters-Klassen nicht berücksichtigt wird. Das bedeutet, für jede Region-Klasse, für jede Topic-Klasse und für jede Industry-Klasse wird ein Verzeichnis angelegt.
Jedes Suffix Array eines jeden Dokuments wird in jedem Verzeichnis gespeichert, das die Klassen repräsentiert, denen das Dokument zugeordnet ist. Das geschieht pro Teilmenge der Trainingsdaten.

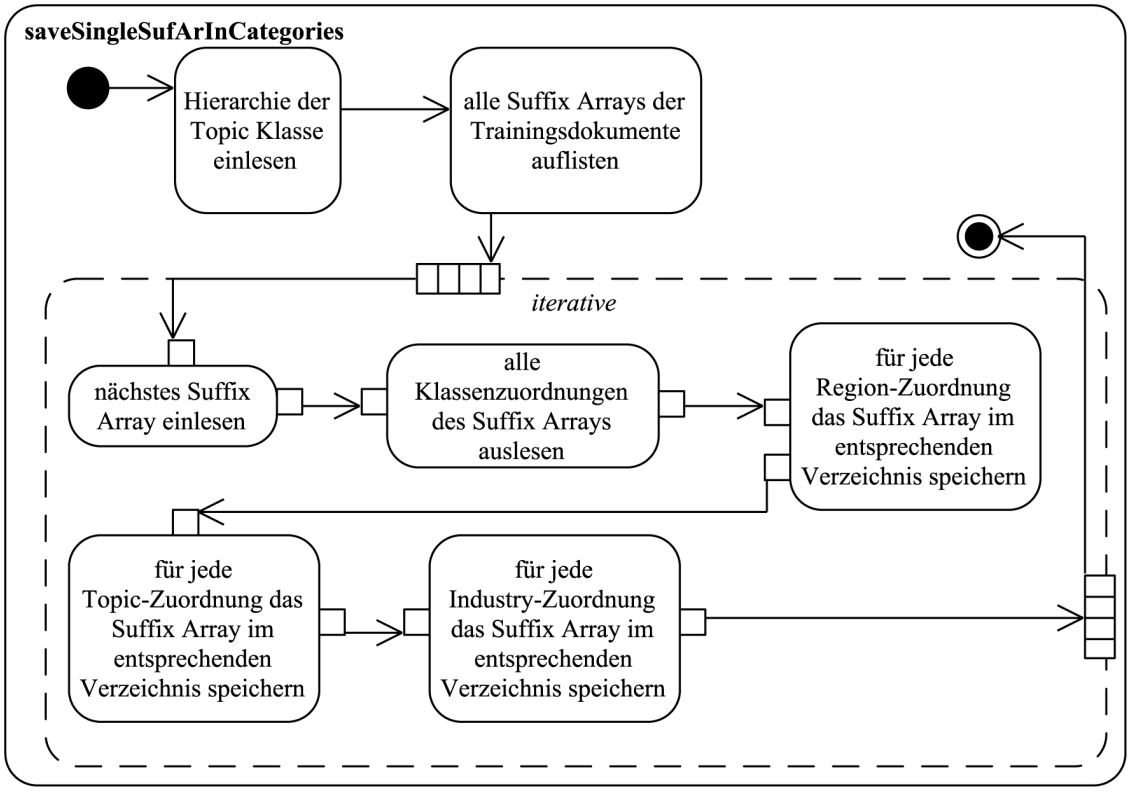


Abbildung 4.23: Ablauf der Operation `saveSingleSufArInCategories`

4.1.7.4 Generalisierte Suffix Arrays erstellen

Für jedes der oben beschriebenen Verzeichnisse mit den einzelnen Suffix Arrays wird ein generalisiertes Suffix Array, wie in Kapitel 3.1.5.3 erläutert, erstellt. Es handelt sich um ein generalisiertes Suffix Array, in dem alle Dokumente der Reuters-Klasse, repräsentiert durch ihre einzelnen Suffix Arrays, enthalten sind.

4.1.7.5 Text-Suffix-Fragment-Features ermitteln für das Klassifizieren mit Text-Suffix-Fragment-Features

Mit Hilfe der generalisierten Suffix Arrays werden die TSF-Features, die während des Klassifizierens verwendet werden, um die Klassenzugehörigkeit des zu klassifizierenden Dokuments zu bestimmen, ermittelt. Das Verfahren wurde in Kapitel 3.2.3.1 erläutert. Pro Reuters-Klasse wird eine bestimmte Anzahl an TSF-Features generiert, die in jeweils einer Datei pro Klasse abgespeichert werden. Das Speichern erfolgt als serialisiertes Objekt der Java-Klasse `Hashtable`, in der die Java-Klasse `String`, also das TSF-Feature, den Schlüssel bildet und die Java-Klasse `PhraseData` den Wert.

4.1.7.6 Zusammenfassen der Text-Suffix-Fragment-Features

Um bei der späteren Erstellung der Vektoren eindeutige TSF-Features zur Verfügung zu haben, muss pro Teilmenge *eine* Datei mit allen TSF-Features aller Klassen erstellt werden. Das vermeidet die mehrfache Betrachtung des gleichen Features.
Aus diesem Grund werden in der Operation `joinFeaturesSubsets`, zu sehen in Abbildung 4.24, alle Feature-Objekte der einzelnen Reuters-Klassen eingelesen und in einem Java-Objekt der Java-Klasse `Hashtable` vereinigt. Durch die Festlegung des TSF-Features als Schlüssel der Hashtabelle kann jedes TSF-Feature nur ein einziges Mal dort vorhanden sein. Der Wert, also das Objekt der Klasse `PhraseData`, muss dagegen aktualisiert werden. Das geschieht, indem die dort vorhandenen Daten zusammengeführt werden. Abschließend wird das Objekt mit allen Features in einer Datei abgelegt.

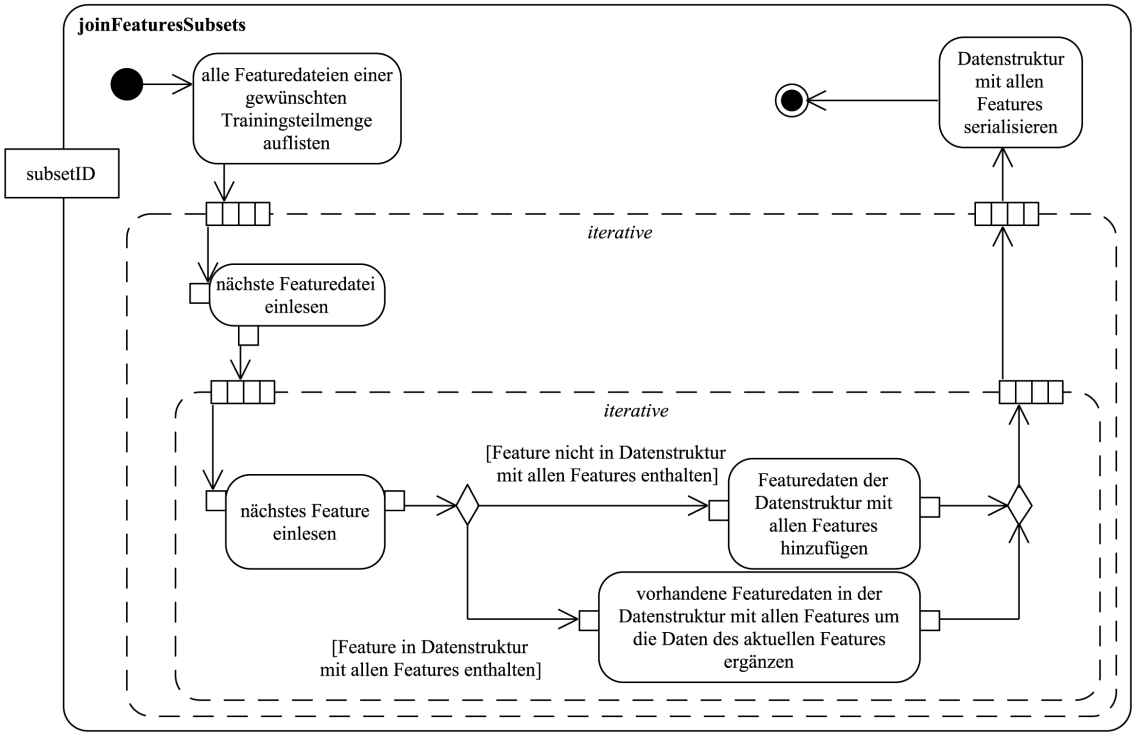


Abbildung 4.24: Ablauf der Operation `joinFeaturesSubsets`

4.1.7.7 Indizieren der Text-Suffix-Fragment-Features für das Klassifizieren mit Text-Suffix-Fragment-Features

Zur Erzeugung der Feature-Vektoren der Dokumente müssen die Dokumente daraufhin überprüft werden, ob die ermittelten Features in ihnen enthalten sind. Um

diesen Vorgang zu erleichtern, werden die ermittelten Features innerhalb der Operation `generateIndexes`, zu sehen in Abbildung 4.25 auf S. 255, indiziert. Das bedeutet, es werden zwei zusätzliche Datenstrukturen erzeugt. Die erste speichert alle in den Features auftretenden Anfangszeichen und einen Zeiger auf die Position des ersten TSF-Features mit diesem Anfangszeichen. Die zweite enthält alle auftretenden ersten drei Zeichen sowie einen Zeiger auf die Position des ersten TSF-Features mit diesen drei ersten Zeichen und einen Zeiger auf die Position des letzten TSF-Features mit diesen drei Zeichen.
Mit Hilfe dieser zusätzlichen Datenstrukturen ist die Suche nach TSF-Features vereinfacht worden, da sie ein „Springen“ innerhalb der Menge der TSF-Features ermöglichen.[1] Diese Datenstrukturen werden gemeinsam mit allen Features in einer Datei als serialisiertes Java-Objekt gespeichert.

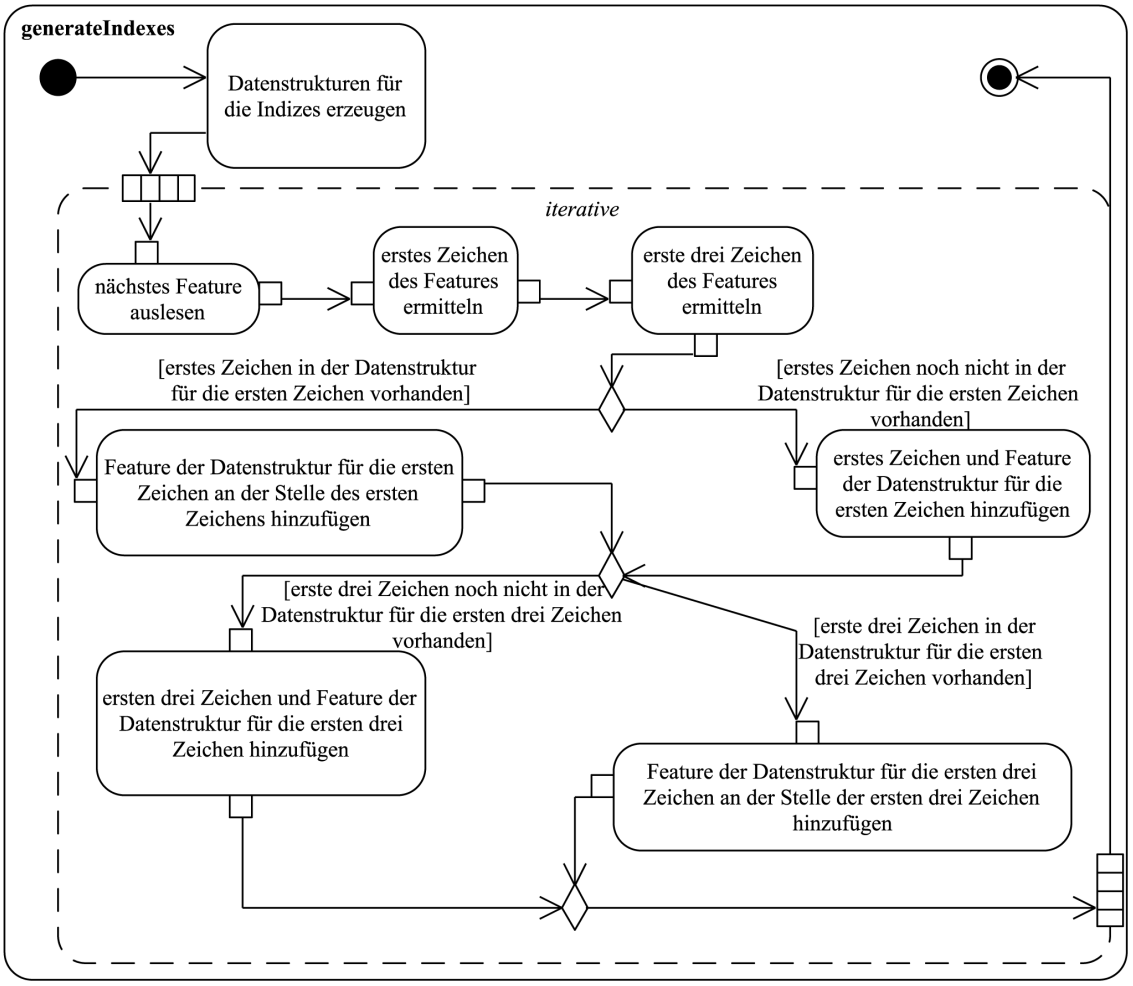


Abbildung 4.25: Ablauf der Operation `generateIndexes`

1 Vgl. Navarro u.a. (2000), S. 218.

4.1.7.8 Erzeugen der Text-Suffix-Fragment-Feature-Vektoren für die Trainingsdaten

Unter Zuhilfenahme des Index auf den TSF-Features und der Suffix Arrays der einzelnen Dokumente ist es nun möglich, für jedes Dokument einen Vektor zu erstellen. Jeder Dokumentenvektor zeigt an, welche TSF-Features aller Trainingsdokumente im Dokument enthalten sind. In der vorliegenden Arbeit werden die IDs dieser TSF-Features in einem Vektor gespeichert, da nur dargestellt wird, welche TSF-Features enthalten sind. Die nicht-enthaltenen TSF-Features werden nicht explizit gespeichert.

Bei der Erstellung des Vektors für ein Trainingsdokument macht man sich das Vorhandensein des Supraindex über die ersten drei Zeichen aller TSF-Features und den Supraindex über die ersten drei Zeichen des Suffix Arrays des Dokuments zu Nutze. Das bedeutet, der Supraindex über die ersten drei Zeichen des Suffix Arrays des Dokuments wird durchlaufen und für jeweils einen Eintrag, also für jeweils drei im Suffix Array und damit im Dokument auftretende drei Anfangszeichen wird überprüft, ob TSF-Features existieren, die ebenfalls mit diesen drei Zeichen beginnen. Dazu muss nur im entsprechenden Supraindex für die Features nachgeschaut werden. Sind Features mit diesen drei Anfangszeichen vorhanden, so werden sie alle im Bereich des Suffix Arrays, der durch die drei Anfangszeichen begrenzt ist, gesucht.[1] Die Suche nach einem TSF-Feature im Suffix Array ist durch den Bereich mit den drei Zeichen als Anfangszeichen begrenzt. Das bedeutet, das erste Suffix, welches betrachtet wird, ist das, auf das der Startzeiger des Index zeigt. Das letzte Suffix des Bereichs ist entsprechend das, worauf der Endezeiger für diesen Bereich zeigt. Solange sich die aktuelle Position, an der nach dem Suffix gesucht wird, zwischen diesen beiden Positionen des Suffix Arrays befindet und das gesuchte TSF-Feature noch nicht gefunden wurde, wird weitergesucht. Die Suche besteht aus einem Textvergleich zwischen dem Suffix an der aktuellen Position im Suffix Array und dem TSF-Feature. Sind beide gleich, so ist das TSF-Feature enthalten und die Suche kann abgebrochen werden. Das Gleiche trifft zu, wenn das Suffix im Suffix Array mit dem TSF-Feature beginnt. Ansonsten wird die nächste Position im Suffix Array überprüft, bis die Endposition des Bereichs mit den drei Zeichen erreicht wird. Wurde eine Übereinstimmung gefunden, so wird die ID des gefundenen TSF-Features im

1 Anmerkung: Die Reihenfolge, die Anfangszeichen des Suffix Arrays zu betrachten und dann die entsprechenden TSF-Features, sofern sie in der Menge der TSF-Features vorhanden sind, im Suffix Array zu suchen, ist deshalb so gewählt worden, weil potenziell eine große Menge an TSF-Features über alle Dokumente gefunden wurde, jedoch nur eine Teilmenge davon in den einzelnen Dokumenten vorhanden ist. Eine umgekehrte Reihenfolge, also das Suchen aller TSF-Features in jedem Dokument, würde den Aufwand erhöhen.

Vektor für das Dokument gespeichert. Danach wird das nächste TSF-Feature, das mit den drei Zeichen beginnt, betrachtet.
Als Algorithmus zur Erstellung eines Vektors für ein Dokument der Trainingsdaten ergibt sich der folgende:

Algorithmus 23 erstelleVektorFuerTrainingsdokument

Eingabe: Suffix Array SA_{d_a} für das Dokument d_a mit Supraindex, alle TSF-Features mit Supraindex

Ausgabe: Vektor $\vec{f}_{d_a}$ für das Dokument d_a

```
1: for alle drei Zeichen in Supraindex von SA_{d_a} do
2:     Betrachte nächste drei Zeichen, das ist dZ
3:     Ermittle alle Features mit dZ als Anfangszeichen, das ist features
4:     if features ≠ ∅ then
5:         Erzeuge einen Iterator über die Suffixe von SA_{d_a}, die mit dZ
           beginnen, das ist iterator
6:         while iterator hat Elemente do
7:             Lies das nächste Element in features aus, das ist feature
8:             Bestimme Start- und Endposition von dZ in SA_{d_a}, das sind
               start und end
9:             isInArray = false
10:            while (start ≤ end) und isInArray == false do
11:                Lies das Suffix an start aus, das ist suffix
12:                if suffix == feature then
13:                    isInArray = true
14:                else if suffix beginnt mit feature then
15:                    isInArray = true
16:                else
17:                    start = start + 1
18:                end if
19:            end while
20:            if isInArray == true then
21:                Schreibe die ID von feature in den Vektor f⃗_{d_a}
22:            end if
23:        end while
24:    end if
25: end for
```

Für die Implementierung ergibt sich der beschriebene Ablauf. D.h., die Operation `createVectorsSingleSufSubsetsNew` wird aufgerufen und darin erfolgt die Erstellung aller Vektoren der Trainingsdaten. Zu sehen sind die Operation sowie alle anderen dazugehörenden Operationen bei der Beschreibung der Erstellung der Vektoren für die Testdaten in den Abbildungen 4.29 bis 4.32 auf den Seiten 264 bis 265.

4.1.7.9 Erzeugen der Text-Suffix-Fragment-Feature-Datenobjekte für die Trainingsdaten

4.1.7.9.1 Erzeugen der TSF-Feature-Datenobjekte für die Trainingsdaten für die Naive-Bayes-, Decision-Tree- und k-Nearest-Neighbour-Algorithmen

Prinzipiell sind alle Informationen über die Dokumente der Trainingsdaten bereits vorhanden, jedoch an mehreren Stellen verteilt. Aus diesem Grund erfolgt eine Erzeugung von einem Datenobjekt für jedes Dokument, das alle Informationen enthält, die zum Klassifizieren benötigt werden. Zusätzlich werden auch die Informationen über die Reuters-Klassen, also die klassenbezogenen Informationen, ermittelt und zusammengefasst.
Dazu wird in der Operation `prepareCategorizationSubsets`, zu sehen in Abbildung 4.26 auf S. 259, für jede Reuters-Klasse ein Objekt der Java-Klasse `Category` erzeugt, das eine eindeutige ID für die Reuters-Klasse, deren Namen, die Häufigkeit des Vorkommens der einzelnen Features in dieser Reuters-Klasse und die Anzahl der Trainingsdokumente, die zu dieser Reuters-Klasse gehören, enthält. Diese Informationen werden für jede Reuters-Klasse pro Teilmenge erzeugt und entsprechend als serialisiertes Java-Objekt abgespeichert.
Für jedes Dokument der Teilmenge wird in der Operation `processFileSubset`, siehe Abbildung 4.27 auf S. 260, ein Objekt der Klasse `Data` erzeugt. Dieses enthält die eindeutige ID des Dokuments, seinen Vektor und alle Reuters-Klassen, zu denen das Dokument gehört, aufgeteilt in die drei Klassenfamilien Region, Industry und Topic. Auch diese Java-Objekte werden serialisiert in Dateien abgelegt, damit auf sie während des Klassifizierens zurückgegriffen werden kann.

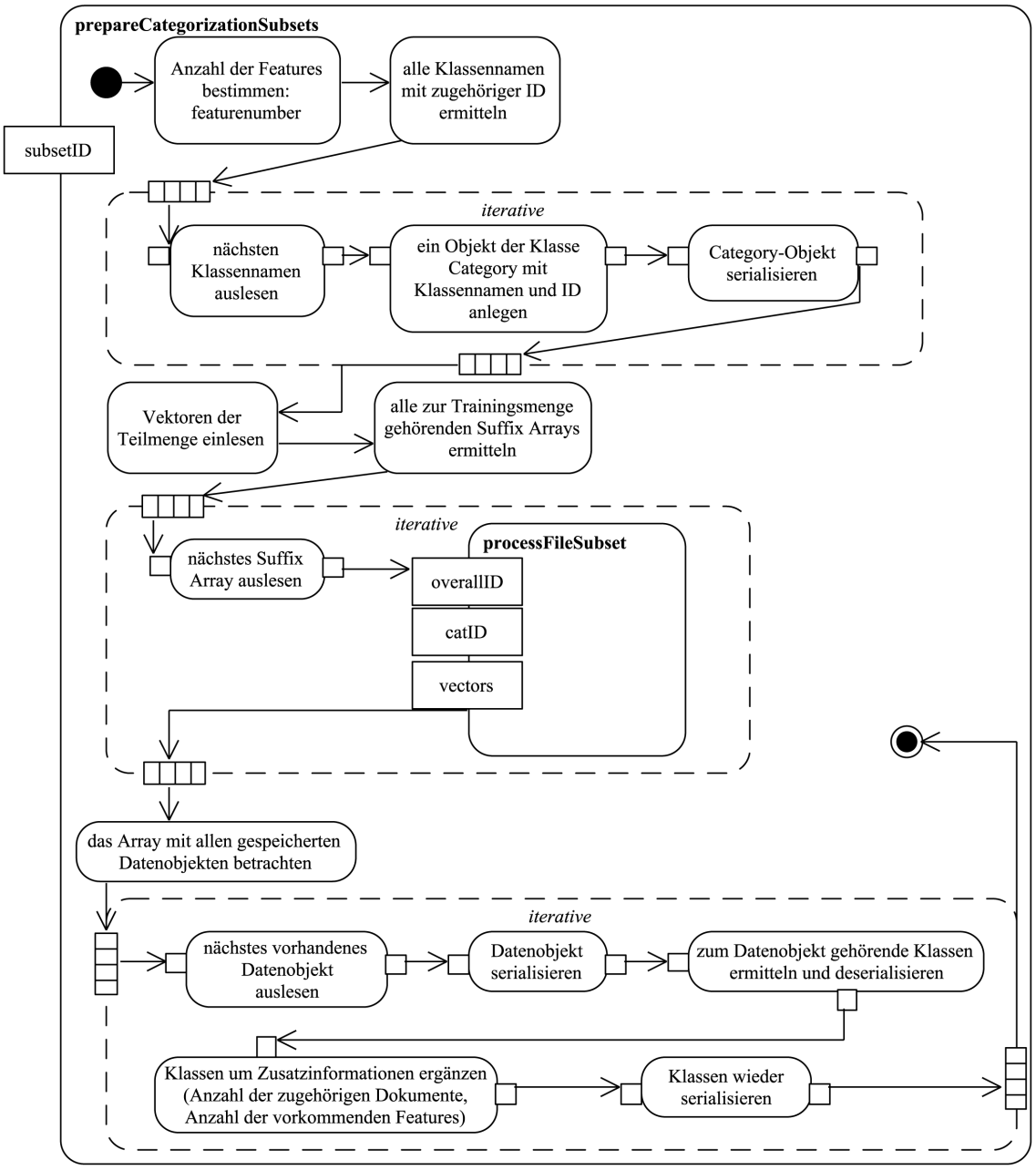


Abbildung 4.26: Ablauf der Operation `prepareCategorizationSubsets`

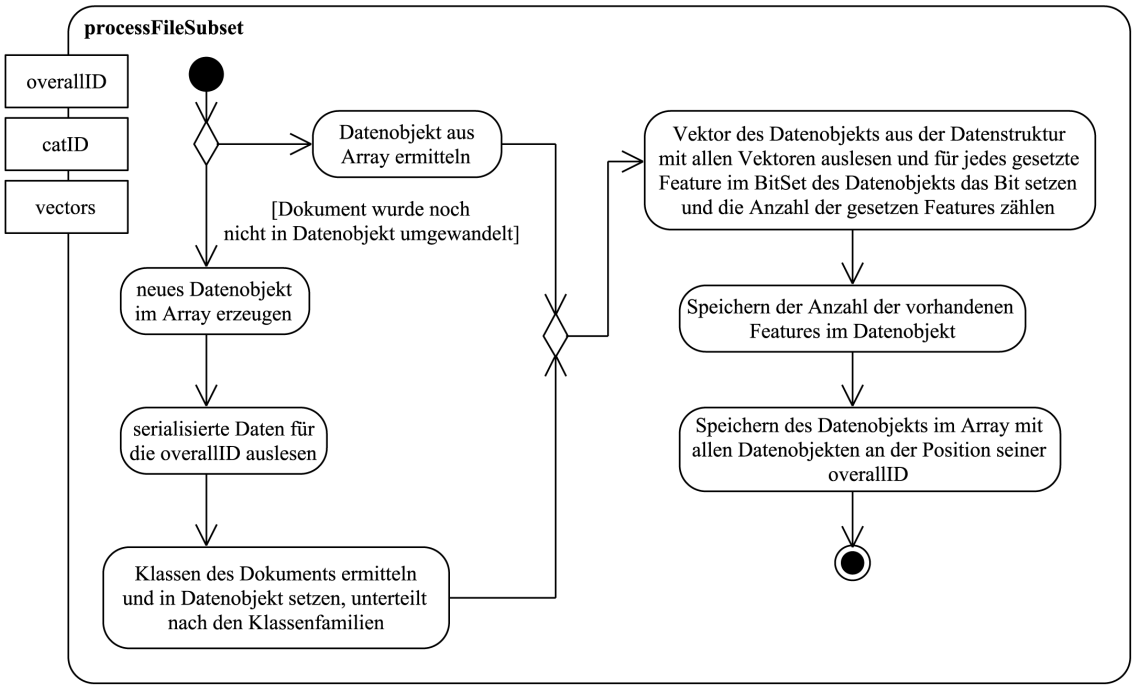


Abbildung 4.27: Ablauf der Operation `processFileSubset`

4.1.7.9.2 Erzeugen der TSF-Feature-Datenobjekte für die Trainingsdaten für den Support-Vector-Machine-Algorithmus

Da das Klassifizieren durch den Support-Vector-Machine-Algorithmus mit einer Software durchgeführt wird, die nicht selbst implementiert wurde[1], unterscheidet sich das Erzeugen der Datenobjekte für die SVM von der Erzeugung der Datenobjekte für die anderen Algorithmen.
Für die SVM müssen die Vektoren der Dokumente der Trainingsdaten in das Eingabeformat der Software umgewandelt werden. Dazu wird die Operation `prepareCategorizationSubsetsSVMMulticlass` aufgerufen. Sie liest die erzeugte Vektordatei mit allen Dokumenten der Trainingsteilmenge der gewünschten Klassenfamilie ein. Zusätzlich wird über alle Suffix Arrays der Dokumente iteriert. Das dient dazu, die Klasse, zu der das jeweilige Dokument gehört, auszulesen. Die ID der Klasse wird in einer Datei gespeichert, gefolgt von einem Leerzeichen und allen Feature-IDs des entsprechenden Dokuments aus der Vektordatei. Nach jeder Feature-ID steht ein Doppelpunkt und eine „1“. Anstatt der Eins kann auch ein Gewicht angegeben werden. Wichtig ist, dass die Feature-IDs aufsteigend angegeben werden, das ist in der Implementierung der Fall. Getrennt werden die Feature-IDs durch ein Leerzeichen.

1 Diese Software wird in Kapitel 4.1.9.2.4 auf S. 289 dieser Arbeit beschrieben

Um später die Evaluation durchführen zu können, muss rekonstruierbar sein, welcher Vektor zu welchem Dokument gehört. Da nur die Klassen-ID und die Feature-IDs für die SVM benötigt werden, wird eine zusätzliche Datei erzeugt, die die overallID[1] des Dokuments und die dazugehörige Zeilennummer der SVM-Vektordatei speichert. Abschließend wird sowohl die SVM-Vektordatei als auch die Mappingdatei gespeichert. Dieser Ablauf ist in Abbildung 4.28 zu sehen.

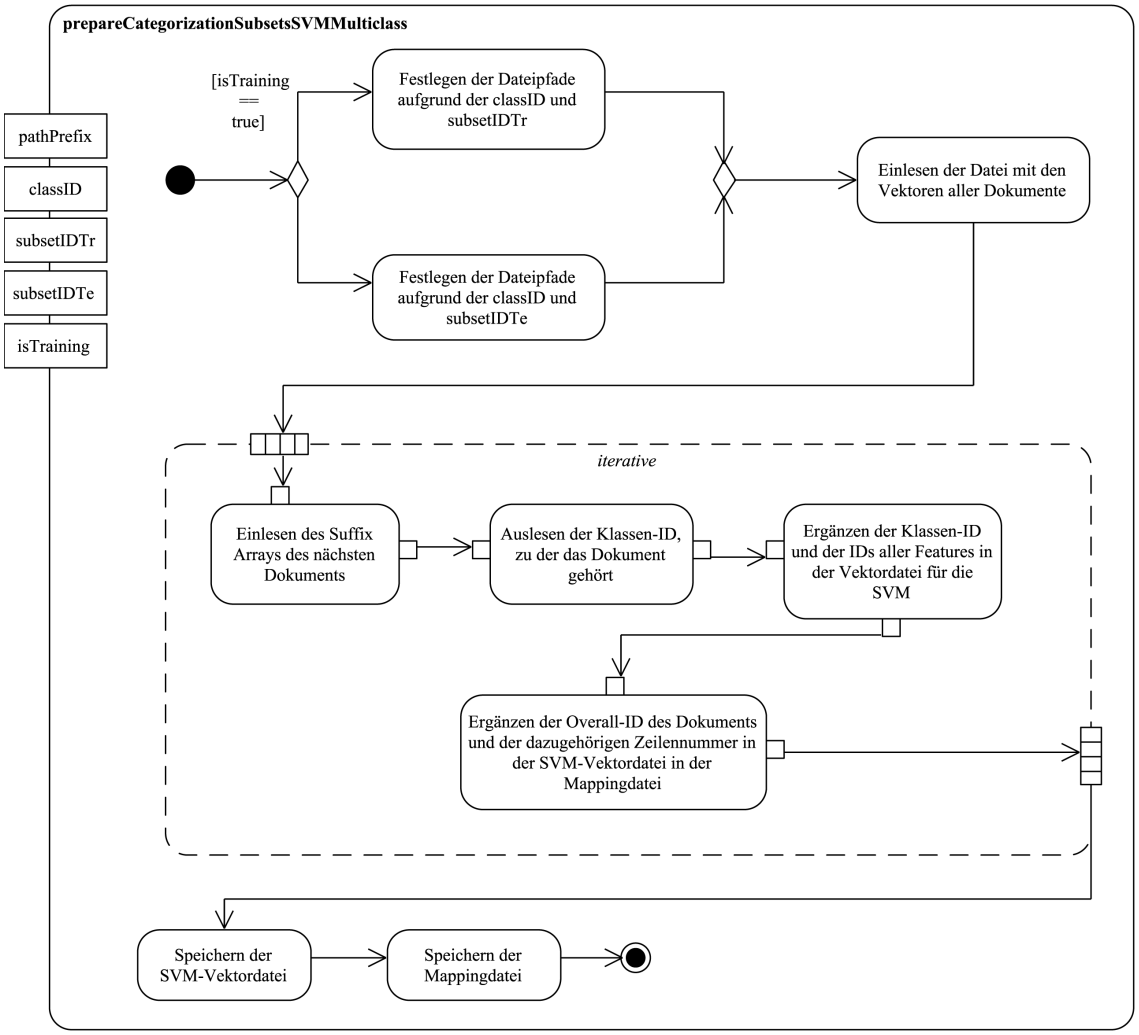


Abbildung 4.28: Ablauf der Operation `prepareCategorizationSubsetsSVMMulticlass`

1 Dabei handelt es sich um eine systemweit eindeutige ID, die von der Verfasserin pro Dokument vergeben wurde.

4.1.8 Testdaten vorbereiten für das Klassifizieren mit Text-Suffix-Fragment-Features

4.1.8.1 Test-Teilmengen erstellen für das Klassifizieren mit Text-Suffix-Fragment-Features

Die vorhandenen 781.265 Testdaten werden in Teilmengen aufgeteilt, um die Durchführungsgeschwindigkeit der verschiedenen Experimente zu erhöhen. Die genauen Parameter, also wie viele Teilmengen erstellt werden, wie viele Dokumente diese jeweils umfassen und welche Anzahl an Klassenzugehörigkeiten die ausgewählten Dokumente haben, können den Beschreibungen der jeweiligen Experimente entnommen werden, siehe Kapitel 4.3.2. Die Teilmengen sind durch ein Verzeichnis symbolisiert, in dem sich die ausgewählten und konvertierten Reuters-Daten, die zur Teilmenge gehören, befinden. Sie sind als serialisierte Java-Objekte der Java-Klasse `FileToCategorize` vorhanden.

4.1.8.2 Suffix Arrays für die Testdaten erstellen

Für jede dieser Teilmengen der Testdaten, die auf die zuvor beschriebene Weise erstellt worden sind, wird für jedes enthaltene und durch das `FileToCategorize`-Objekt repräsentierte Dokument ein Suffix Array erzeugt. Die Erzeugung erfolgt, wie in Kapitel 3.1.4.4 beschrieben. Diese erzeugten Suffix Arrays werden als serialisierte Java-Objekte der Java-Klasse `SuffixArrayToSave` in Bezug auf die Teilmenge, zu der sie gehören, gespeichert.

4.1.8.3 Erzeugen der Text-Suffix-Fragment-Feature-Vektoren für die Testdaten

Für jedes Testdokument einer Testteilmenge wird ein Feature-Vektor, bezogen auf eine Trainingsteilmenge, erstellt. Der Bezug zur jeweiligen Trainingsteilmenge muss vorhanden sein, da in jeder Trainingsteilmenge andere Dokumente enthalten sind und aus diesem Grund auch andere TSF-Features ermittelt wurden. Die Vektoren der Testdokumente werden erstellt, indem überprüft wird, welche der ermittelten TSF-Features in diesen Testdokumenten enthalten sind - für diese TSF-Features wird die ID des TSF-Features im Vektor gespeichert - und welche nicht. Das bedeutet, man erhält für jedes Testdokument einen anderen Vektor in Abhängigkeit von der betrachteten Trainingsteilmenge.
Um die TSF-Features in den Testdokumenten zu suchen, benutzt man in der Operation `createVectorsSingleSufSubsetsNew`, zu sehen in Abbildung 4.29 bis 4.33 auf

S. 264 bis 266, wie zuvor bei der Erstellung der Vektoren für die Trainingsdokumente, das jeweilige Suffix Array des Testdokuments und die indizierten TSF-Features der Trainingsteilmenge. Durch die Indizierung und die lexikografische Sortierung des Suffix Arrays kann sehr leicht geprüft werden, ob ein bestimmtes TSF-Feature im Suffix Array vorhanden sein kann, und ein „Weiterspringen“ zum nächsten TSF-Feature ist ebenfalls leicht möglich.
Die IDs der im Suffix Array vorhandenen TSF-Features werden im Vektor des zugehörigen Testdokuments gespeichert. Das entspricht in der vorliegenden Arbeit der Markierung, dass das TSF-Feature im zugehörigen Dokument vorhanden ist. Eine explizite Speicherung der nicht vorhandenen TSF-Features im Vektor erfolgt nicht, um diesen nicht unnötig zu vergrößern.

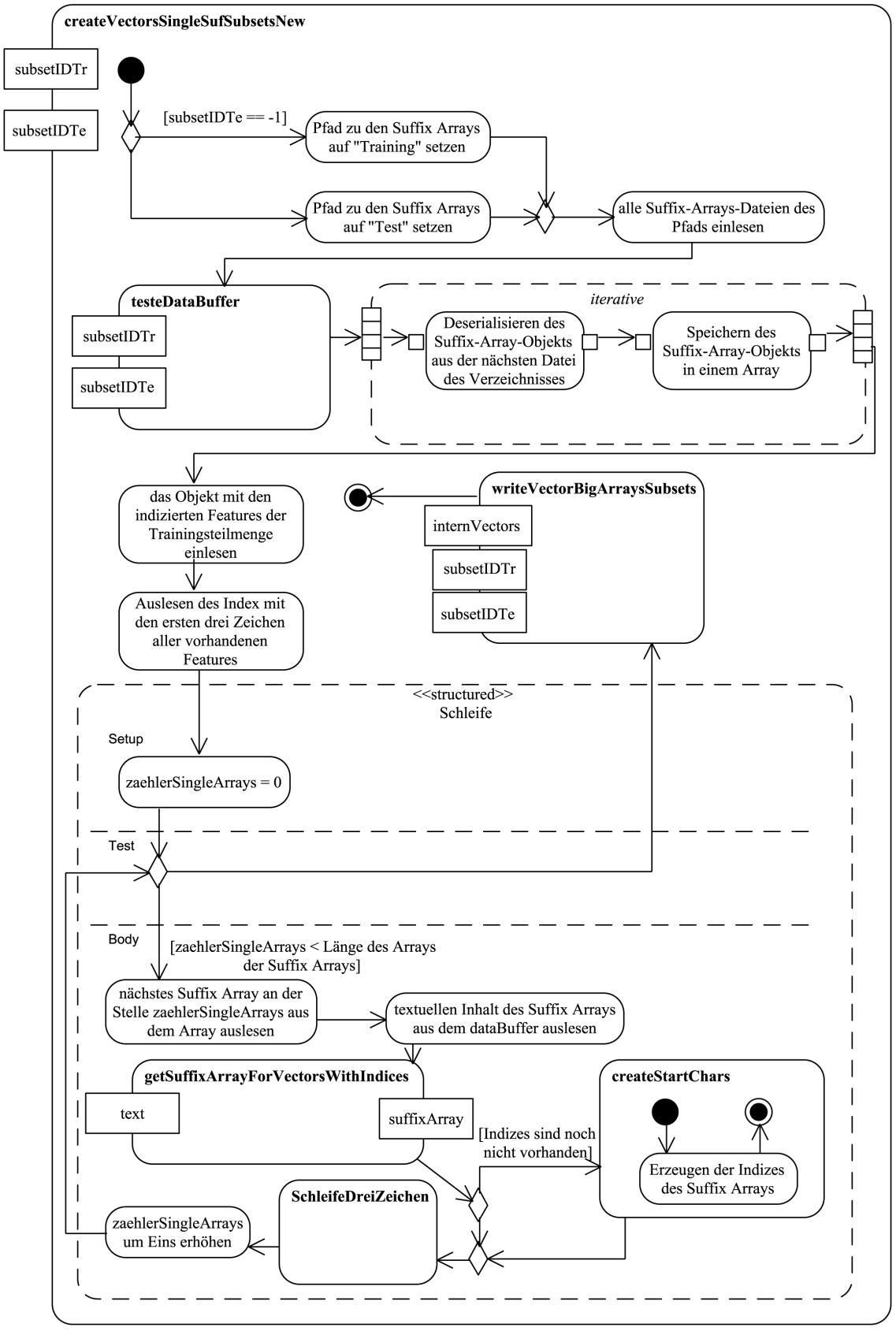


Abbildung 4.29: Ablauf der Operation `createVectorsSingleSufSubsetsNew`

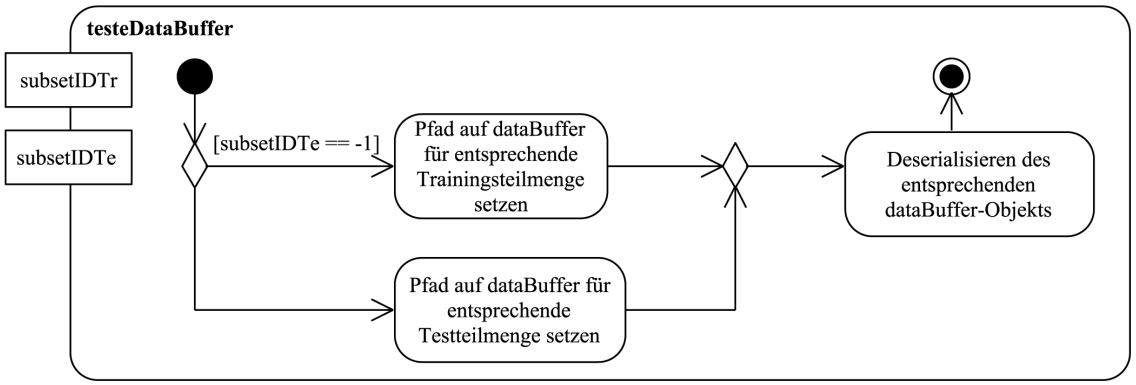


Abbildung 4.30: Ablauf der Operation `testeDataBuffer`

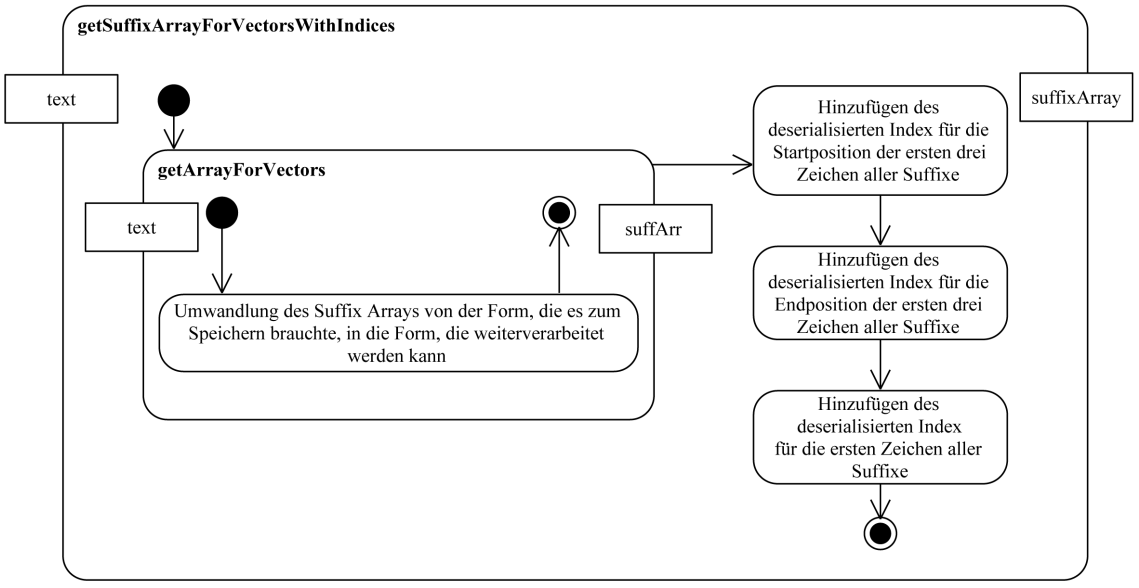


Abbildung 4.31: Ablauf der Operation `getSuffixArrayForVectorsWithIndices`

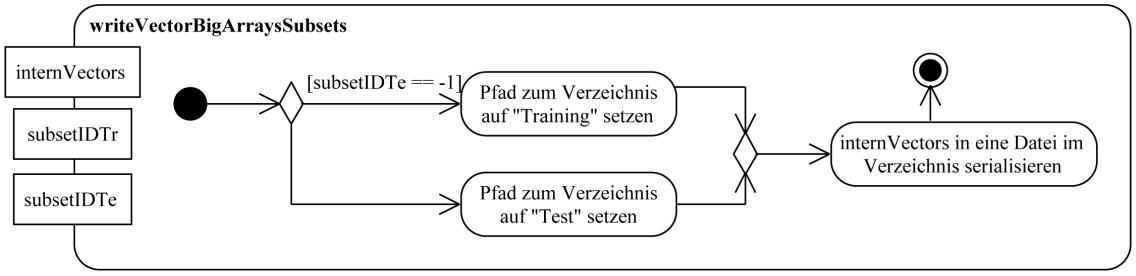


Abbildung 4.32: Ablauf der Operation `writeVectorBigArraysSubsets`

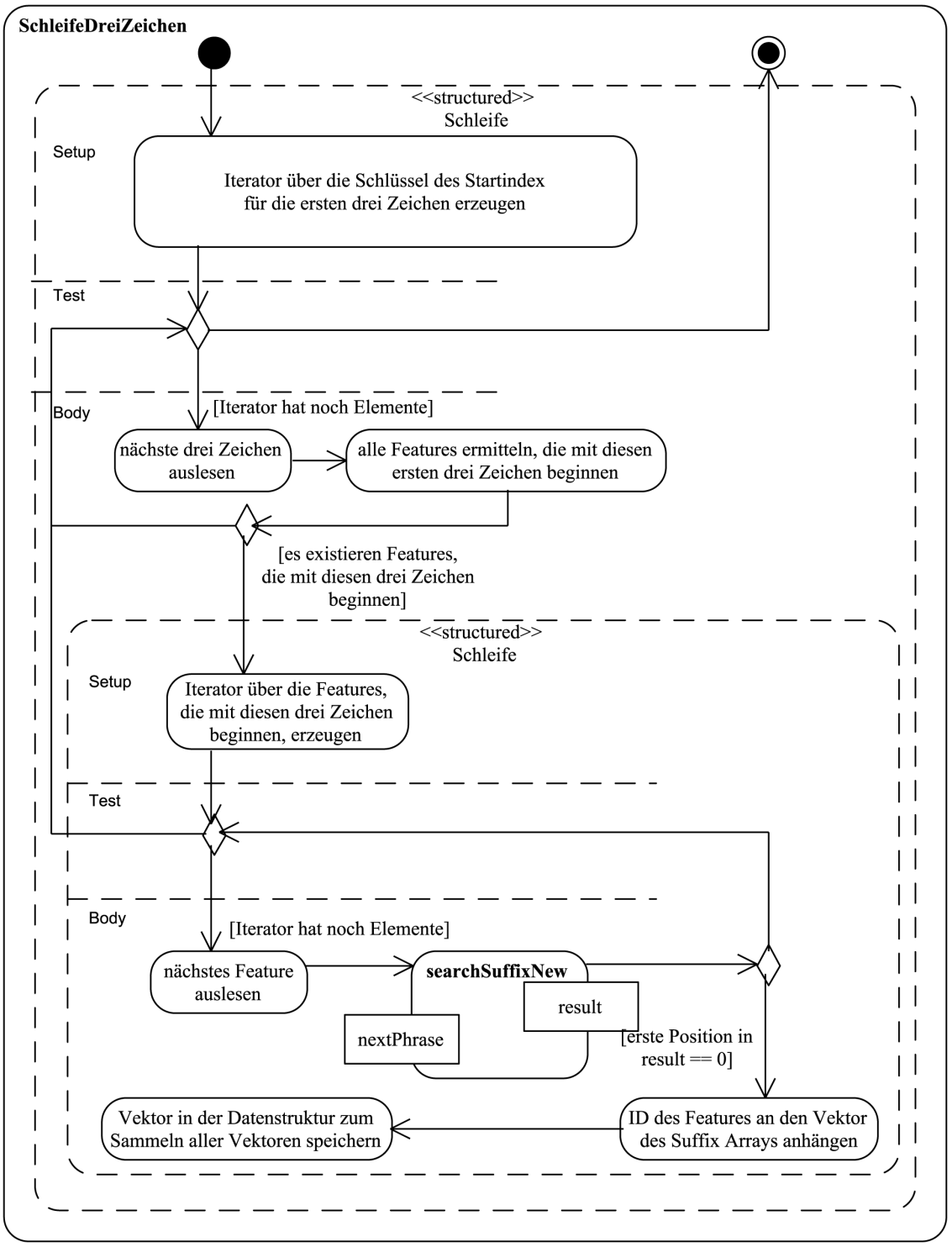


Abbildung 4.33: Darstellung der Schleife über die drei Zeichen

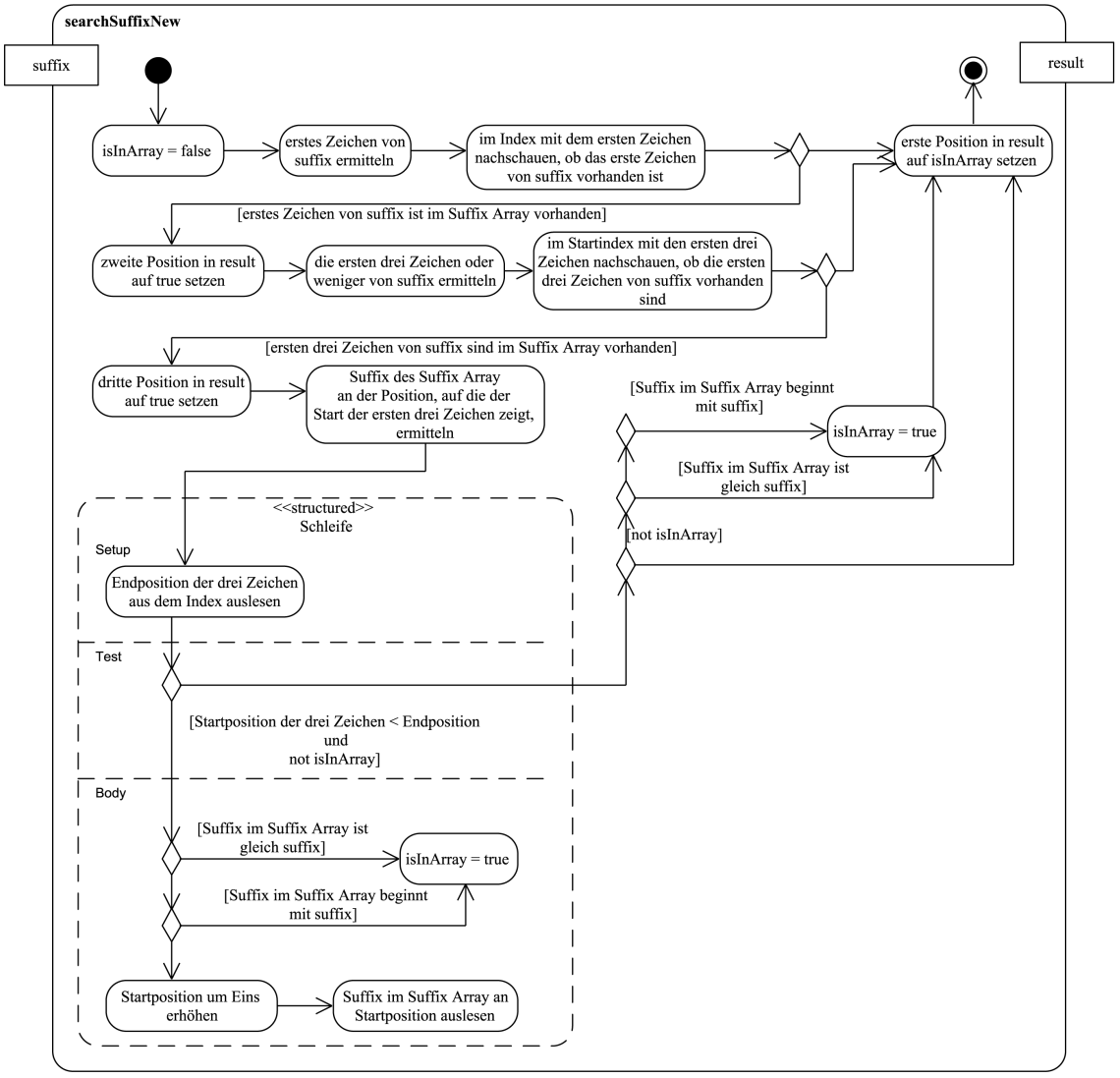


Abbildung 4.34: Ablauf der Operation `searchSuffixNew`

4.1.8.4 Erzeugen der Text-Suffix-Fragment-Feature-Datenobjekte für die Testdaten

4.1.8.4.1 Erzeugen der TSF-Feature-Datenobjekte für die Testdaten für die Naive-Bayes-, Decision-Tree- und k-Nearest-Neighbour-Algorithmen

Prinzipiell sind alle Informationen über die Dokumente der Testdaten bereits vorhanden, jedoch an mehreren Stellen verteilt. Aus diesem Grund erfolgt eine Erzeugung von einem Datenobjekt für jedes Dokument, das alle Informationen enthält, die zum Klassifizieren benötigt werden. Zusätzlich werden auch die Informationen

über die Reuters-Klassen, also die klassenbezogenen Informationen, ermittelt und zusammengefasst.

Das ist nötig, um nach dem Klassifizieren die Evaluation leichter durchführen zu können, ohne die Informationen über die korrekte Klassenzuordnung erst suchen zu müssen. Während des Klassifizierens werden die Klassenzuordnungen nicht berücksichtigt, da nur auf dem Inhalt des Dokuments operiert wird. Das Auslesen und Speichern der Klassenzuordnungen für die Testdokumente erfolgt nach dem gleichen Verfahren wie für die Trainingsdokumente, siehe Kapitel 4.1.7.9.

Um die benötigten Informationen für das Klassifizieren zusammenzufassen, wird in der Operation `processFileSubsetTest`, zu sehen in Abbildung 4.36 auf S. 269, für jedes Dokument der Teilmenge ein Objekt der Klasse `Data` erzeugt. Dieses enthält die eindeutige ID des Dokuments, seinen Vektor und alle Reuters-Klassen, zu denen das Dokument gehört, aufgeteilt in die drei Klassenfamilien Region, Industry und Topic. Auch diese Java-Objekte werden serialisiert in Dateien abgelegt, damit auf sie während des Klassifizierens zurückgegriffen werden kann.

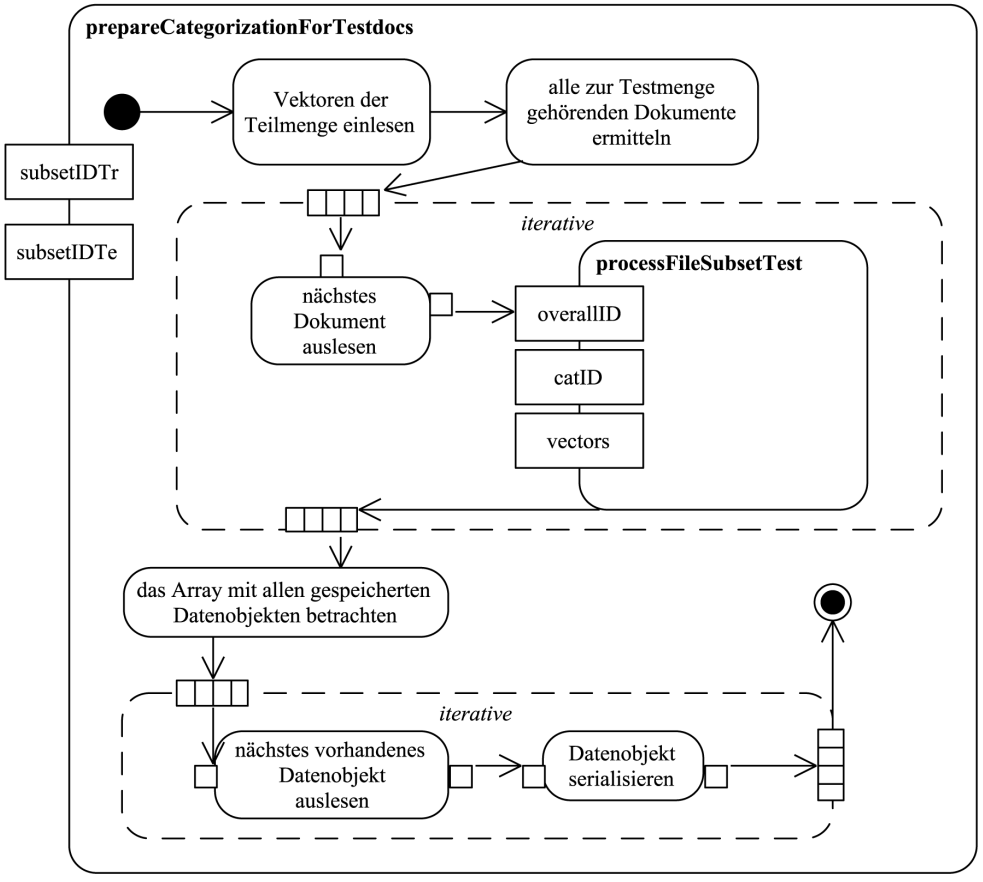


Abbildung 4.35: Ablauf der Operation `prepareCategorizationForTestdocs`

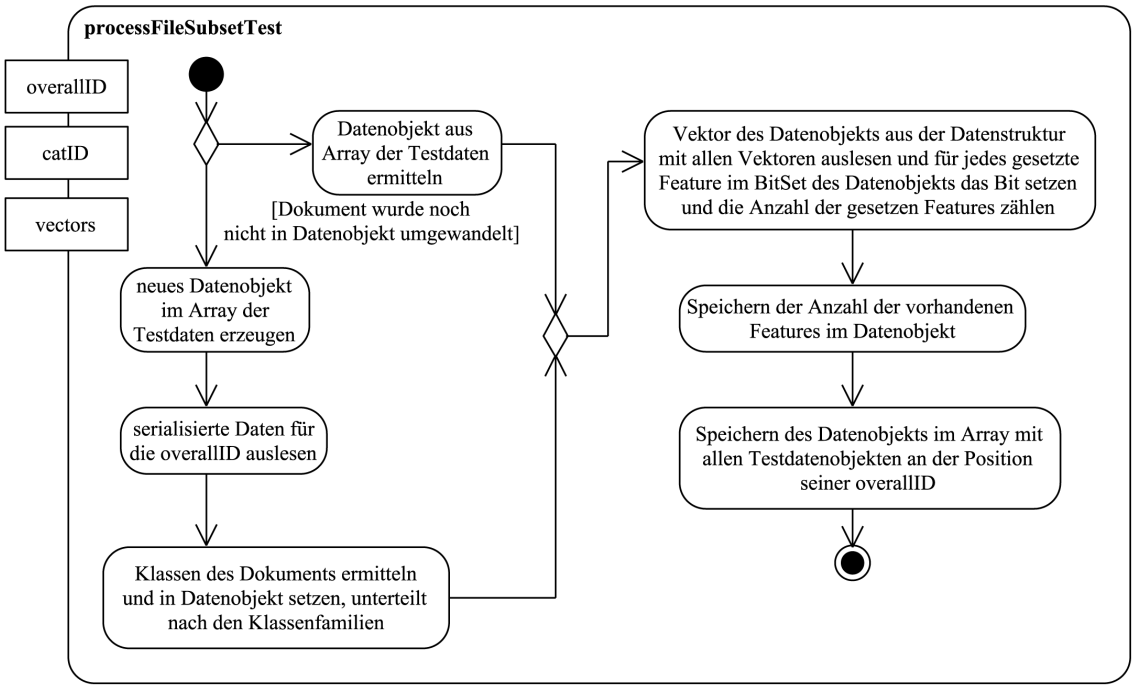


Abbildung 4.36: Ablauf der Operation `processFileSubsetTest`

4.1.8.4.2 Erzeugen der TSF-Feature-Datenobjekte für die Testdaten für den Support-Vector-Machine-Algorithmus

Da das Klassifizieren durch den Support-Vector-Machine-Algorithmus mit einer Software durchgeführt wird, die nicht selbst implementiert wurde[1], unterscheidet sich das Erzeugen der Datenobjekte für die SVM von der Erzeugung der Datenobjekte für die anderen Algorithmen. Genau wie bei den Trainingsdaten wird ebenfalls die Operation `prepareCategorizationSubsetsSVMMulticlass` aufgerufen.
Die Operation führt genau die gleichen Schritte aus wie für die Trainingsdaten, arbeitet jedoch auf den Testdaten und erzeugt für diese die entsprechenden Vektordateien und Mappingdateien für die SVM. Dieser Ablauf ist in Abbildung 4.28 auf S. 261 zu sehen.

1 Diese wird in Kapitel 4.1.9.2.4 auf S. 289 dieser Arbeit beschrieben

4.1.9 Klassifizieren mit Text-Suffix-Fragment-Features

4.1.9.1 Vorbereiten des Klassifizierens mit Text-Suffix-Fragment-Features

4.1.9.1.1 Einlesen der Text-Suffix-Fragment-Feature-Datenobjekte der gewünschten Trainingsteilmenge

Als ersten Schritt für die Durchführung des Klassifizierens müssen die Datenobjekte der gewünschten Trainingsteilmenge eingelesen werden. Dabei werden in der Operation `prepareCategorizationAfterCatSavingSubsets` die serialisierten `Data`-Objekte eingelesen und in einer Speicherstruktur, die alle Trainingsdatenobjekte enthält, abgelegt. Zusätzlich werden weitere Datenstrukturen, die für das Klassifizieren benötigt werden, angelegt. Welche Trainingsteilmenge jeweils eingelesen wird, kann den Beschreibungen der einzelnen Experimente entnommen werden, siehe Kapitel 4.3.2.
Soll mit Hilfe eines Decision Trees klassifiziert werden, so wird der zur gewünschten Trainingsteilmenge gehörende Decision Tree entweder deserialisiert - wenn er zuvor bereits aufgebaut und gespeichert wurde - oder mit Hilfe der Trainingsdatenobjekte erzeugt.[1] Das bedeutet, es erfolgt zunächst eine Überprüfung, ob der gewünschte Decision Tree bereits gespeichert wurde. Ist das der Fall, wird er deserialisiert, so dass er während des Klassifizierens der Testdatenobjekte verwendet werden kann.
Im anderen Fall erfolgt ein Aufbau des Baumes, wie in Kapitel 2.3.3.3 beschrieben und in den Abbildungen 4.37 auf S. 271 bis 4.41 auf S. 273 zu sehen. Die dafür nötigen Dokumente mit ihren Vektoren sind die Datenobjekte der jeweiligen Trainingsteilmenge.
Anschließend wird der Decision Tree innerhalb der Operation `convertDecTreeToDecTreeSave`, zu sehen in Abbildung 4.46 auf S. 277, konvertiert, so dass er serialisiert werden kann, und in einer Datei gespeichert. Abschließend erfolgt in der Operation `evaluateDecTreeTraining`[2], siehe Abbildung 4.38 auf S. 271, eine Evaluation des Decision Trees, indem die Trainingsobjekte mit seiner Hilfe klassifiziert werden. Hierbei muss in allen Fällen das Ergebnis eine 100% korrekte Klassenzuordnung sein, um sicherzustellen, dass der Decision Tree richtig aufgebaut wurde.

1 Bei den anderen verwendeten Methoden zum Klassifizieren sind vorbereitende Schritte beim Einlesen der Datenobjekte nicht nötig.
2 Die Operation `predictCatDecTree` wird an späterer Stelle erläutert und ist in Abbildung 4.55 auf S. 285 zu sehen.

Beim Klassifizieren mit der SVM entfällt das Einlesen der Datenobjekte, da das Klassifizieren durch eine Software durchgeführt wird. Die benötigten Dateien für die Eingabe wurden bereits in früheren Schritten erstellt.

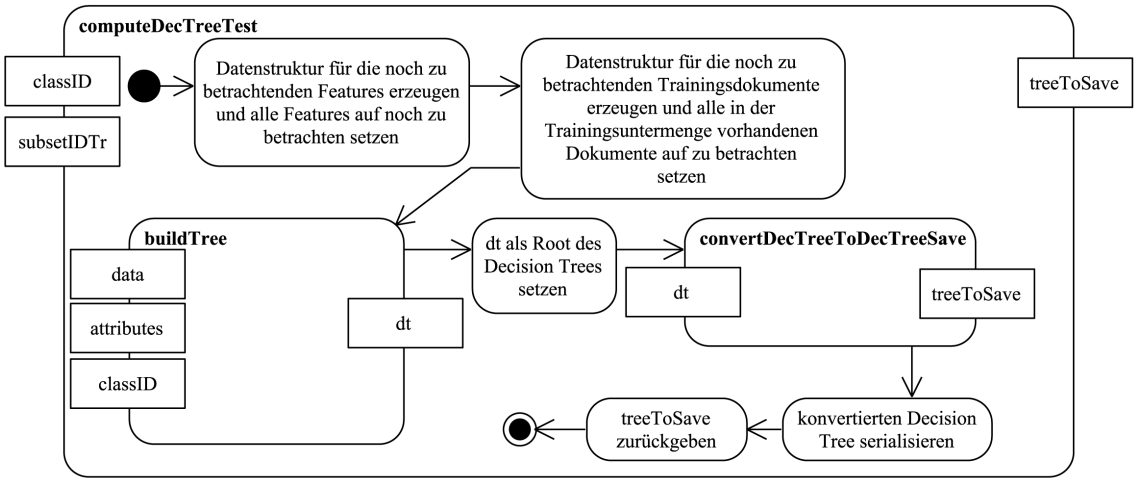


Abbildung 4.37: Ablauf der Operation `computeDecTreeTest`

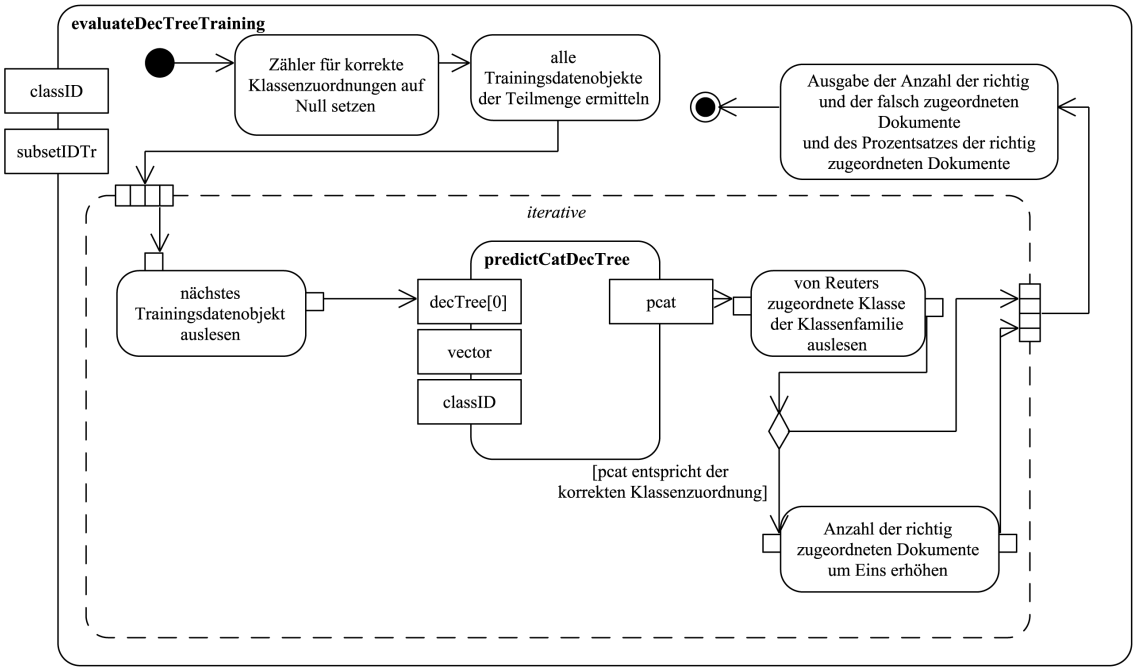


Abbildung 4.38: Ablauf der Operation `evaluateDecTreeTraining`

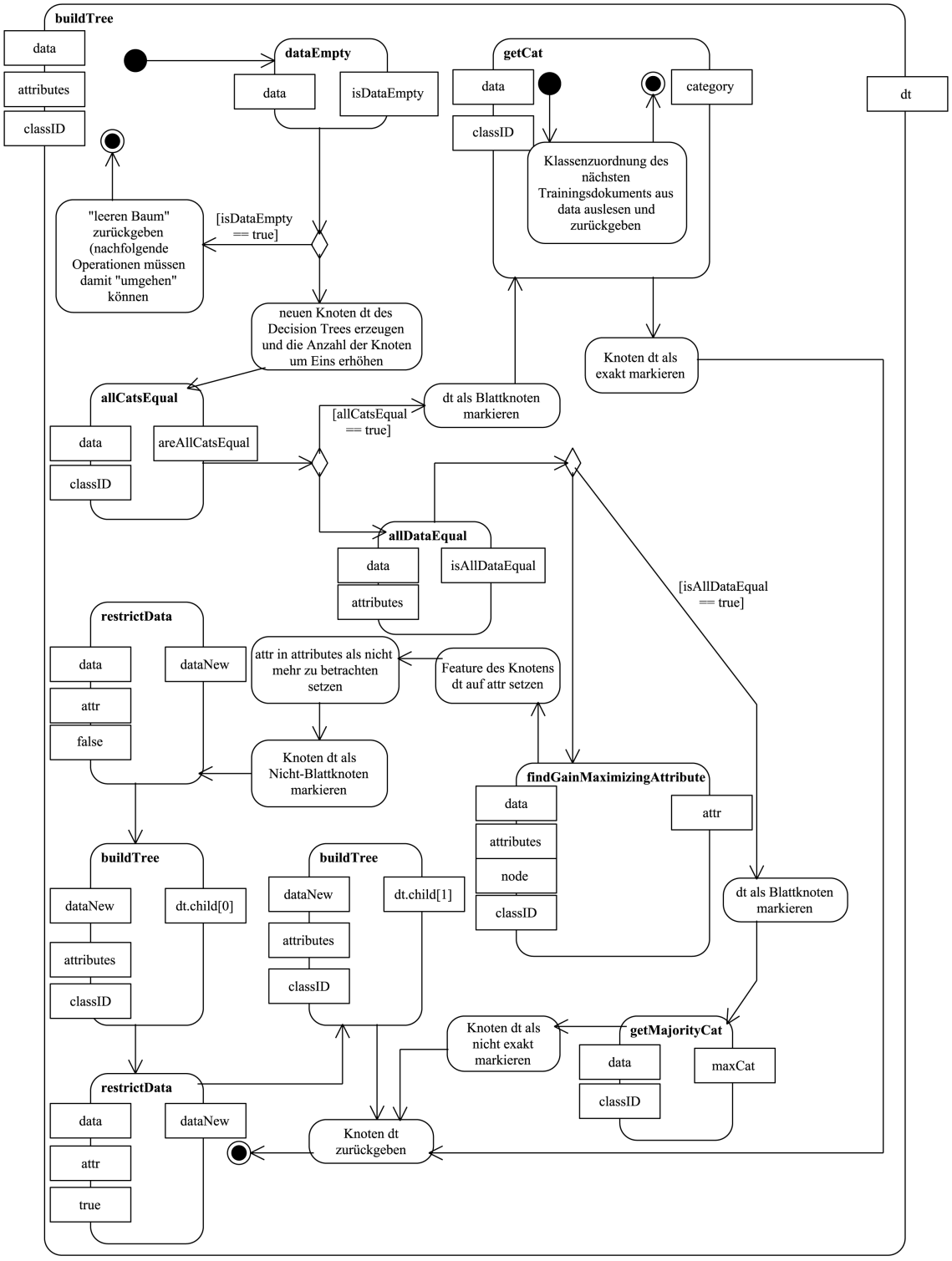


Abbildung 4.39: Ablauf der Operation `buildTree`

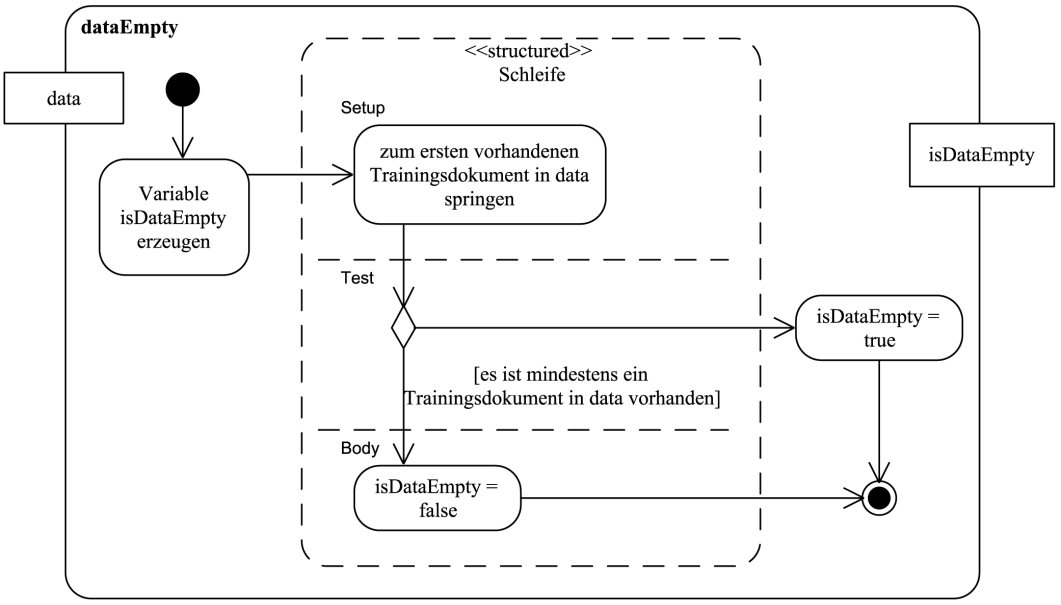


Abbildung 4.40: Ablauf der Operation `dataEmpty`

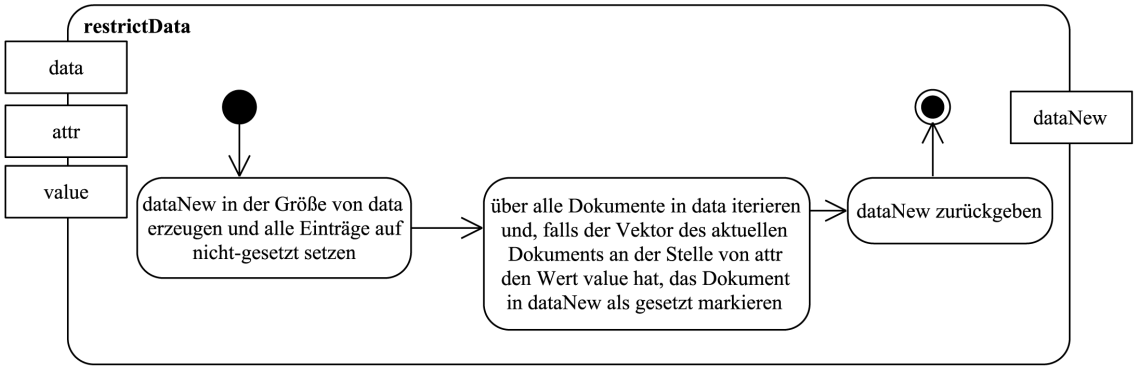


Abbildung 4.41: Ablauf der Operation `restrictData`

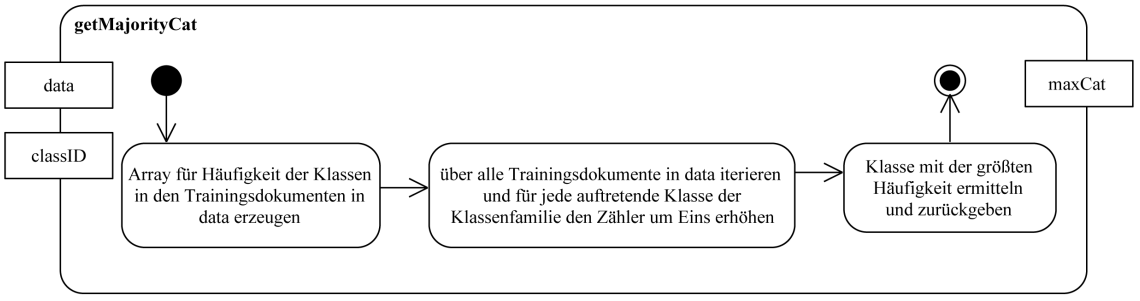


Abbildung 4.42: Ablauf der Operation `getMajorityCat`

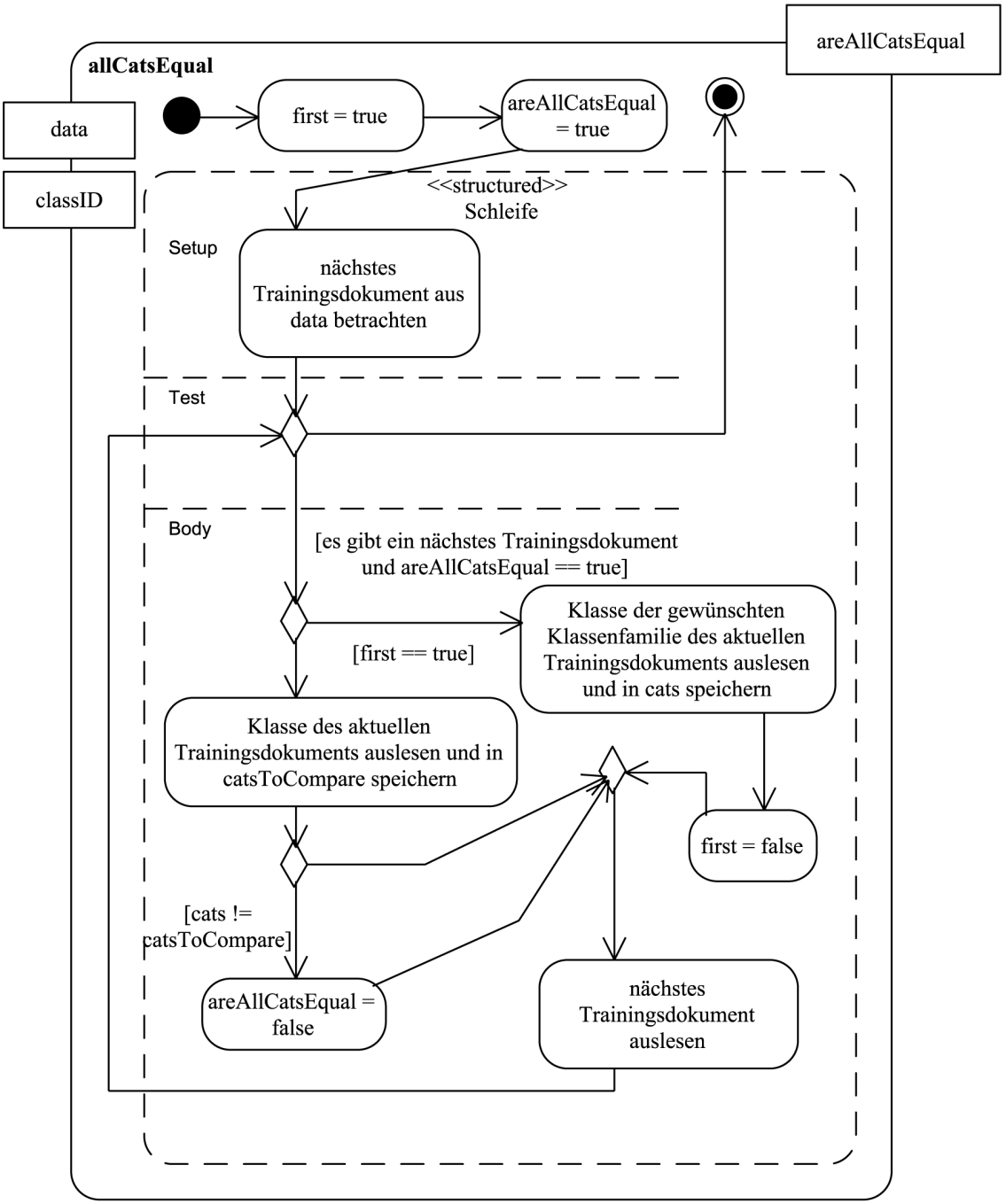


Abbildung 4.43: Ablauf der Operation `allCatsEqual`

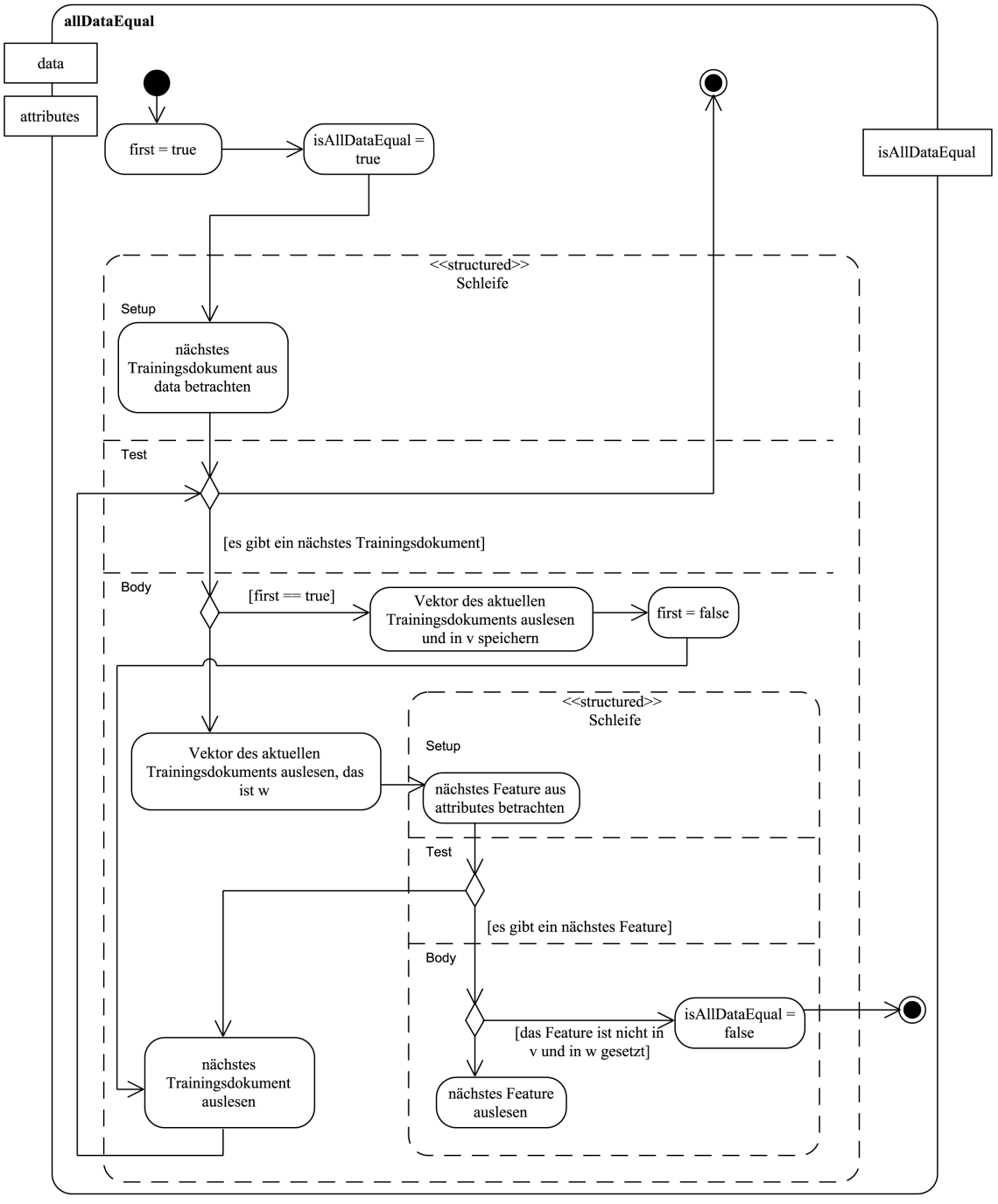


Abbildung 4.44: Ablauf der Operation `allDataEqual`

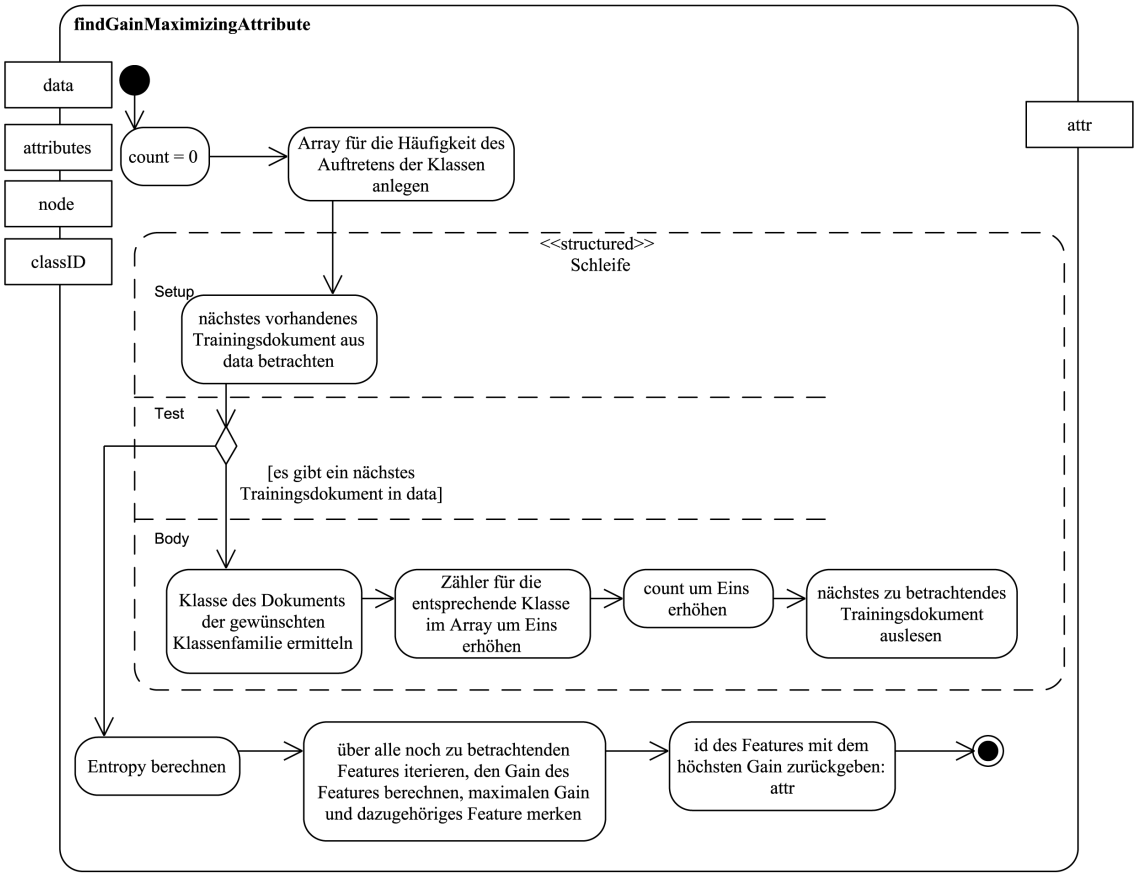


Abbildung 4.45: Ablauf der Operation `findGainMaximizingAttribute`

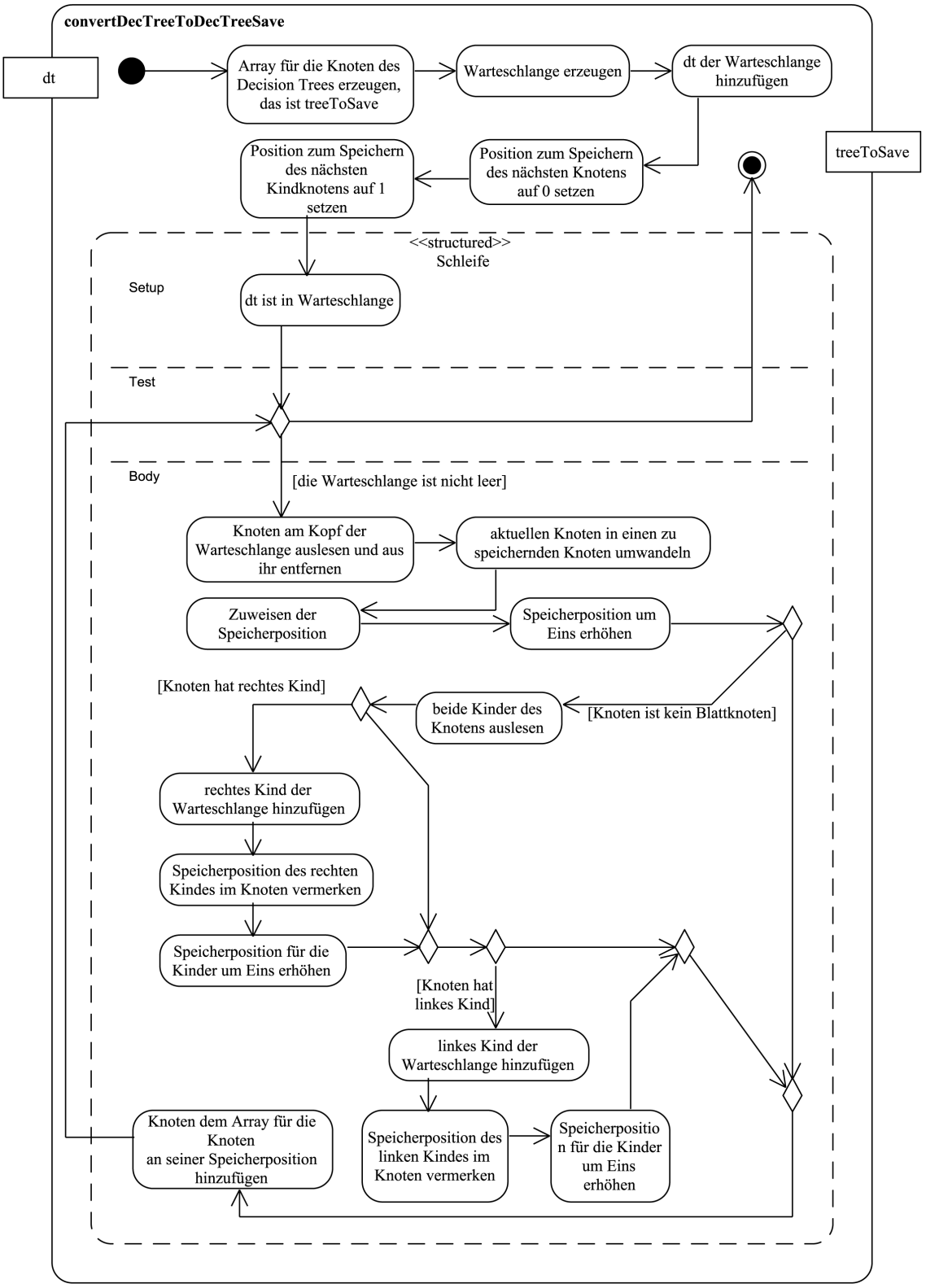


Abbildung 4.46: Ablauf der Operation `convertDecTreeToDecTreeSave`

4.1.9.1.2 Einlesen der Text-Suffix-Fragment-Feature-Datenobjekte der gewünschten Testteilmenge

Die zuvor erstellten Datenobjekte der gewünschten Testdatenteilmenge müssen ebenfalls eingelesen werden. Das geschieht in der Operation `prepareCategorization-AfterDataSavingSubsetTest`, zu sehen in Abbildung 4.47 auf S. 278. Dabei ist darauf zu achten, dass diejenigen Objekte, die für die oben eingelesene Trainingsdatenmenge erstellt wurden, eingelesen werden. Auch hier werden die Testdaten in einer Speicherstruktur abgelegt.
Das Einlesen der Datenobjekte für die Testteilmenge entfällt für die SVM aus den gleichen Gründen, wie beim Einlesen der Trainingsteilmenge erläutert.

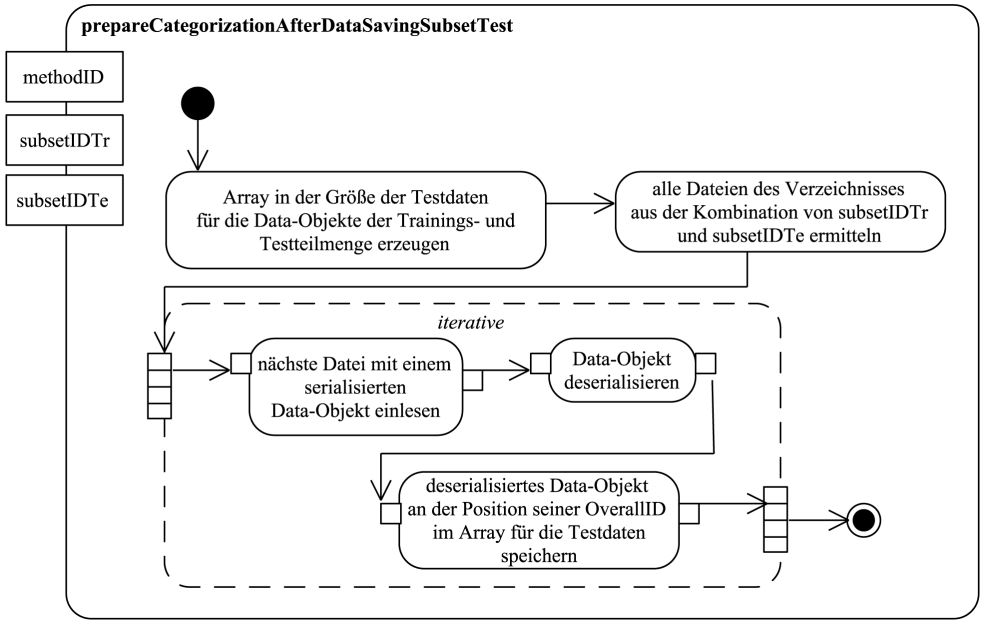


Abbildung 4.47: Ablauf der Operation `prepareCategorizationAfterDataSavingSubsetTest`

4.1.9.2 Durchführen des Klassifizierens mit Text-Suffix-Fragment-Features

4.1.9.2.1 Klassifizieren mit Text-Suffix-Fragment-Features mit dem k-Nearest-Neighbour-Algorithmus

Beim Klassifizieren mit dem k-Nearest-Neighbour-Algorithmus, siehe Kapitel 2.3.3.2, wird zunächst eine Vorverarbeitung mit den Datenobjekten der Trainings- und der Testdatenmenge in der Operation `preprocesskNNTest`, zu sehen in Abbildung 4.48 auf S. 279, durchgeführt. Innerhalb dieser Vorverarbeitung wird die für das Speichern

der paarweisen Cosinuswerte der Vektoren der Trainings- und Testdaten benötigte Datenstruktur angelegt. Zusätzlich wird die Anzahl eines jeden Features über alle Vektoren der Trainingsdaten ermittelt, da diese Werte für die spätere Berechnung des Cosinus benötigt werden.

Während des eigentlichen Klassifizierens wird jedes Testdokument betrachtet. Das geschieht in der Operation `computekNNTest`, zu sehen in Abbildung 4.49 auf S. 281. Zunächst wird der Cosinus zwischen dem Vektor des gerade betrachteten Testdokuments und der Reihe nach jeweils allen Vektoren der Trainingsdokumente berechnet und in der erzeugten Datenstruktur abgelegt. Dadurch ist ein späterer Zugriff auf diese Werte möglich.

Auf diese Berechnung folgt das Speichern derjenigen Trainingsdokumente als nächste Nachbarn des Testdokuments, deren berechneter Cosinuswert am höchsten ist. Dazu werden zunächst die ersten k[1] Trainingsdokumente mit ihren zu dem aktuellen Testdokument berechneten Cosinuswerten gespeichert.

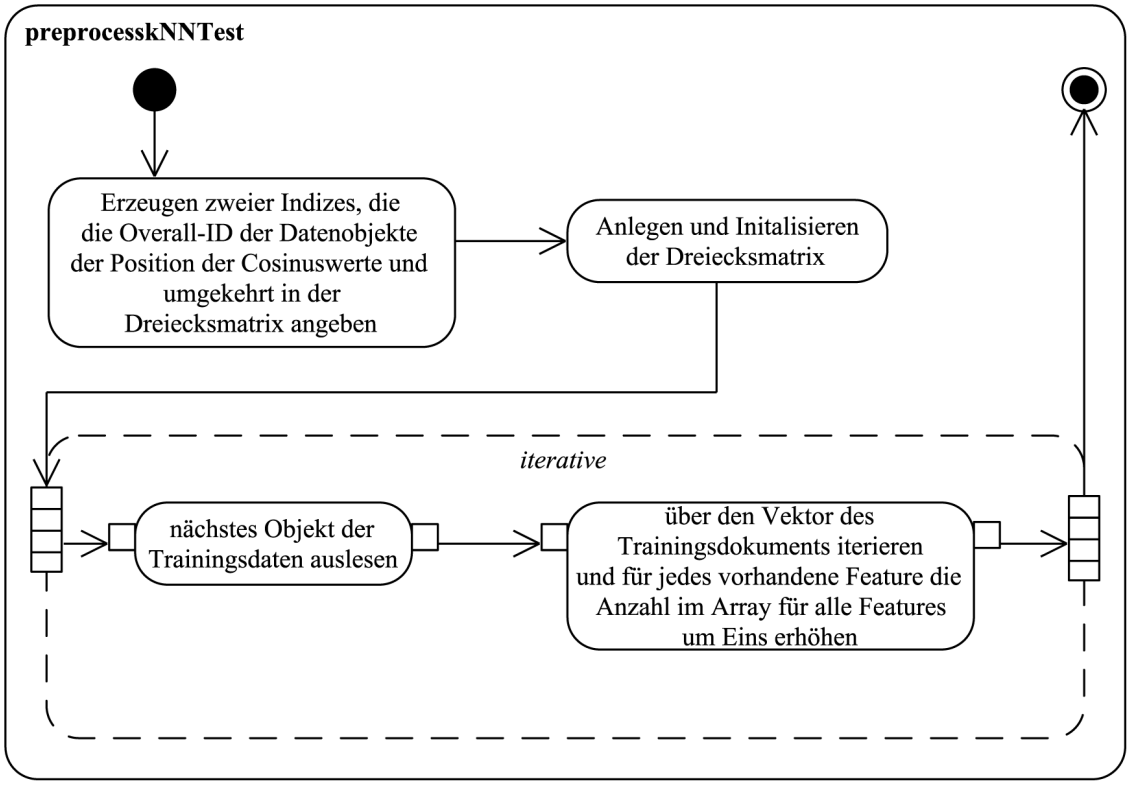


Abbildung 4.48: Ablauf der Operation `preprocesskNNTest`

1 Welchen Wert k einnimmt, welche Trainigs- oder Testteilmengen jeweils verwendet werden und welche Familie von Klassen für die Klassenzuordnung eine Rolle spielt, kann der jeweiligen Experimentbeschreibung, siehe Kapitel 4.3.2, entnommen werden.

Sobald bereits mehr als k Trainingsdokumente betrachtet wurden, erfolgt eine Speicherung des nächsten Trainingsdokuments nur noch, wenn der berechnete Cosinuswert größer als der bisher gespeicherte minimale Cosinuswert ist. Nach dem Durchlaufen aller Trainingsdokumente sind die k Trainingsdokumente mit den höchsten Cosinuswerten in Bezug auf das aktuelle Testdokument gespeichert. Bei diesen Trainingsdokumenten handelt es sich um die k nächsten Nachbarn des aktuellen Testdokuments.

Anschließend wird die Datenstruktur mit den nächsten Nachbarn absteigend nach ihren Cosinuswerten sortiert.[1] Die Vorhersage der Klasse oder der Klassen des aktuellen Testdokuments erfolgt in einem nächsten Schritt.[2]

Zunächst wird die Datenstruktur mit den nächsten Nachbarn durchlaufen. Jeder dort gespeicherte Nachbar hat eine Klassenzuordnung in der betrachteten Familie von Klassen. Für jede so gefundene Klasse wird die Anzahl in einer weiteren Datenstruktur um eins erhöht, so dass nach dem kompletten Durchlaufen aller Nachbarn jede Klassenzuordnung, bezogen auf die Familie von Klassen, dieser Nachbarn erfasst ist.

Anschließend wird die Klasse gesucht, die die höchste Anzahl besitzt. Diese ist, nach dem so genannten Majority-Voting[3], die Klasse des aktuellen Testdokuments.

Die dem aktuellen Testdokument zugeordnete Klasse wird in dem Datenobjekt für das Testdokument abgelegt und das entsprechende Java-Objekt serialisiert, so dass auf alle für die Evaluation benötigten Informationen zugegriffen werden kann.

1 Durch diese Sortierung wird es ermöglicht, die nächsten Nachbarn unterschiedlich zu gewichten, wenn die Klasse oder die Klassen des aktuellen Testdokuments vorhergesagt werden. Das ist in der Implementierung der vorliegenden Arbeit jedoch nicht realisiert worden.

2 In den durchgeführten Experimenten wird jeweils nur eine Klasse für jedes Dokument vom Klassifizierer zugeordnet. Aus diesem Grund wird im Folgenden nur von der Vorhersage *der* Klasse gesprochen. Für die Zuordnung von mehreren Klassen zu einem Dokument siehe Kapitel 2.3.2.2.

3 Majority-Voting bezeichnet einen Vorgang, bei dem in der vorliegenden Anwendung alle benachbarten Trainingsdokumente eine „Stimme“ in der „Wahl“ haben. Das bedeutet, die benachbarten Dokumente sind einer Klasse zugeordnet und geben diese Klasse als ihre „Stimme“ in der „Wahl“ zur richtigen Klasse des Testdokuments an. Die Klasse, die die Mehrheit der „Stimmen“ besitzt, also die höchste Anzahl an benachbarten Trainingsdokumenten, wird die Klasse des Testdokuments, vgl. bspw. Li u.a. (2010), S. 792 f. In der vorliegenden Implementierung wird bei Gleichstand mehrerer Klassen diejenige ausgewählt, welche als erste betrachtet wird, da nur genau eine Klasse zugewiesen wird. Das Verfahren ist sehr weit verbreitet, vgl. bspw. Narasimhamurthy (2005), S. 1988, insbesondere wird es auch häufig im kNN zur Festlegung der Klasse einer Testgröße verwendet, vgl. Paul u.a. (2006), S. 2523. Majority Voting kann auch eingesetzt werden, um mehrere Algorithmen zum Klassifizieren zu verwenden und dann die Klasse über ein Majority Voting über die Ergebnisse der Algorithmen zu bestimmen, vgl. Roja u.a. (2011), S. 358; Paul u.a. (2006), S. 2523. Diese Vorgehensweise wird auch als *Ensemble* von Klassifizierern bezeichnet, vgl. Narasimhamurthy (2005), S. 1993.

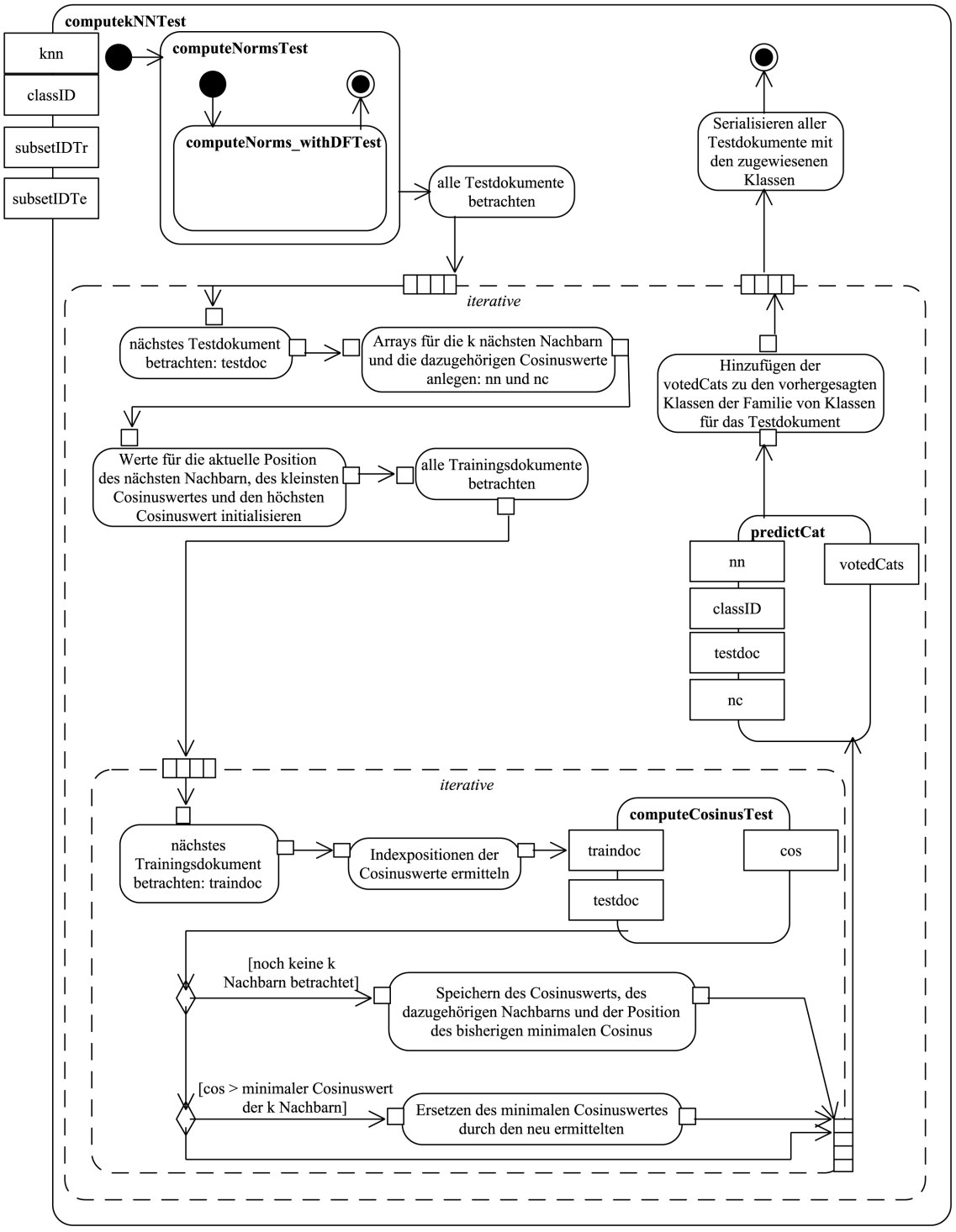


Abbildung 4.49: Ablauf der Operation `computekNNTest`

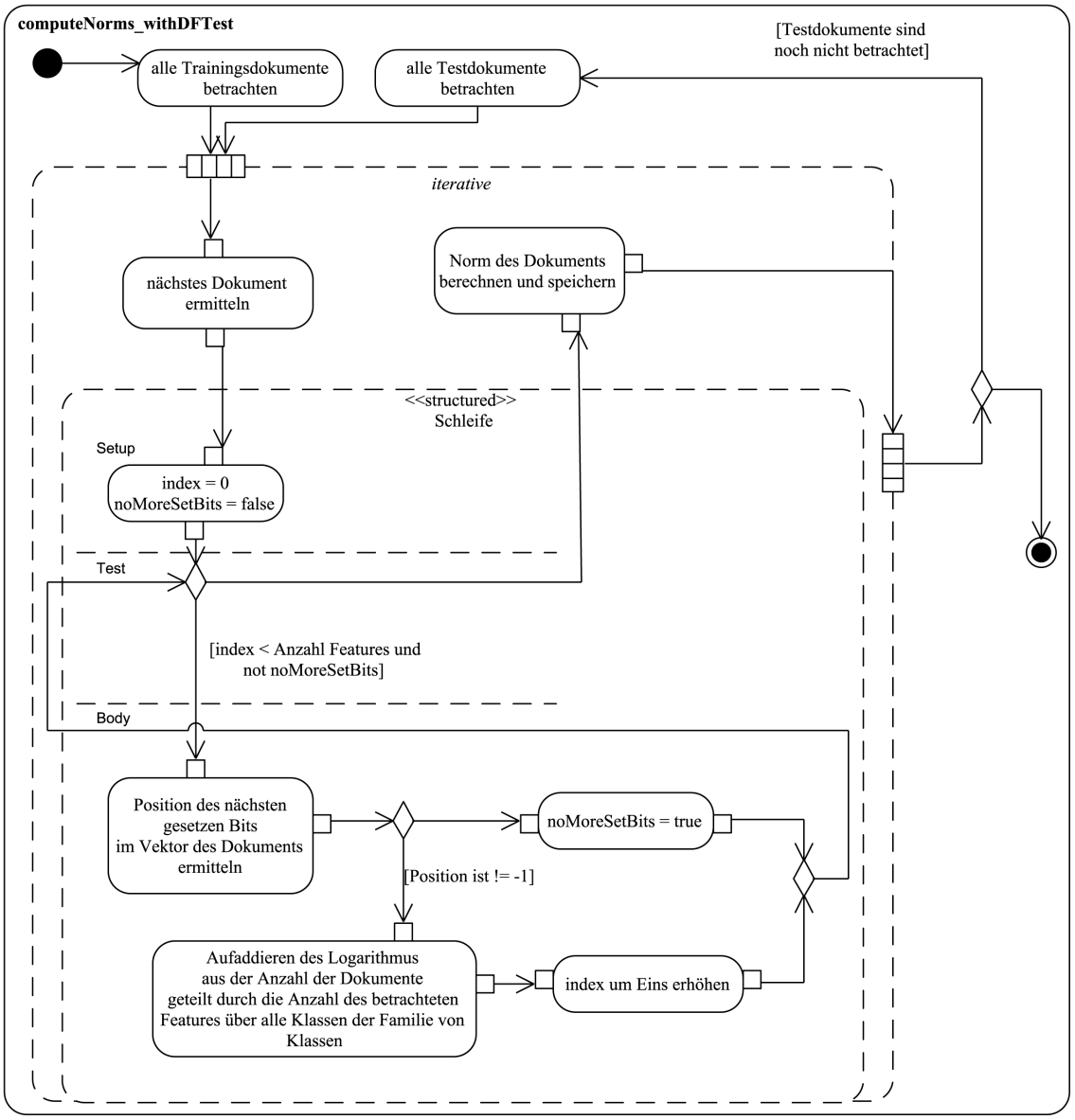


Abbildung 4.50: Ablauf der Operation `computeNorms_withDFTest`

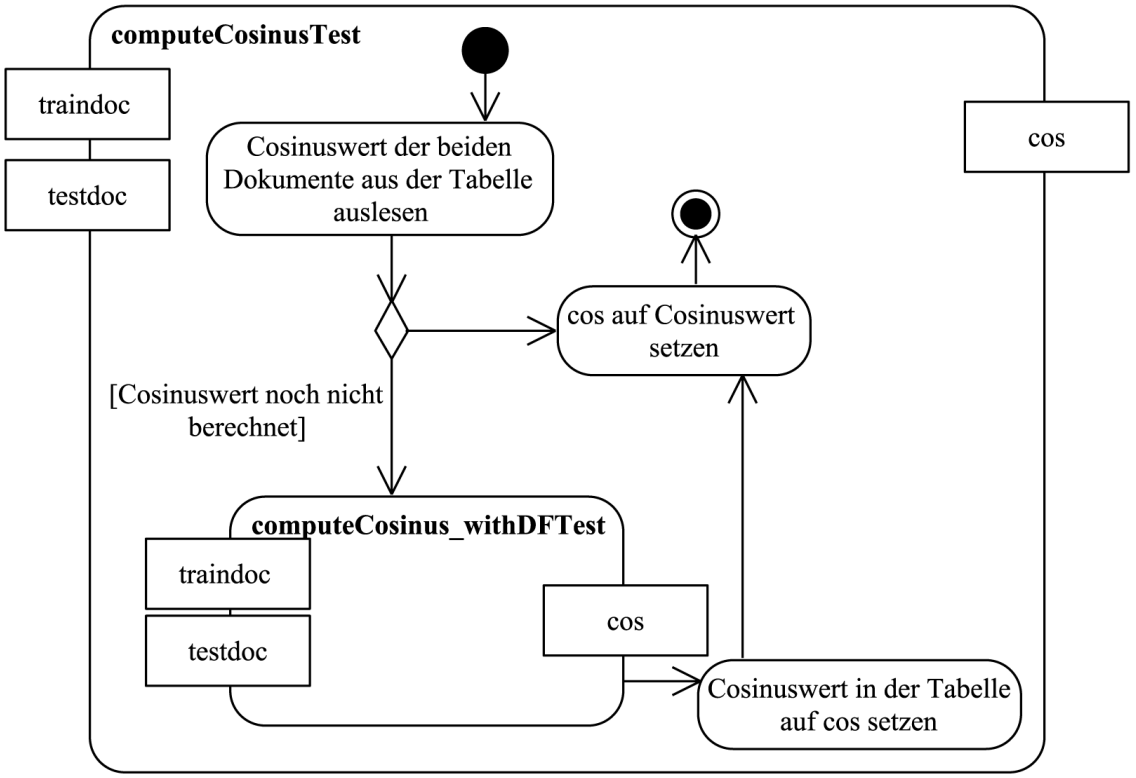


Abbildung 4.51: Ablauf der Operation `computeCosinusTest`

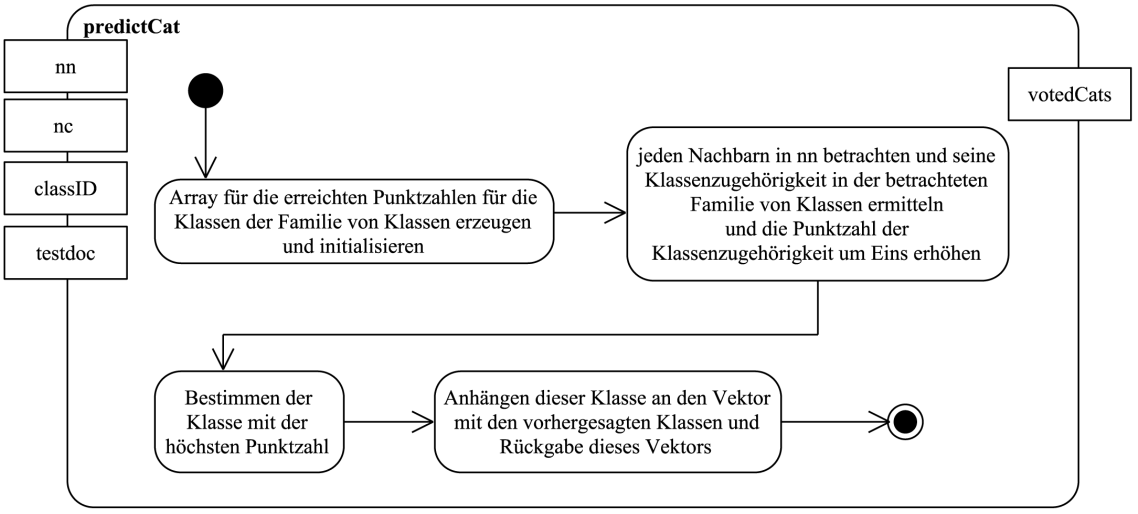


Abbildung 4.52: Ablauf der Operation `predictCat`

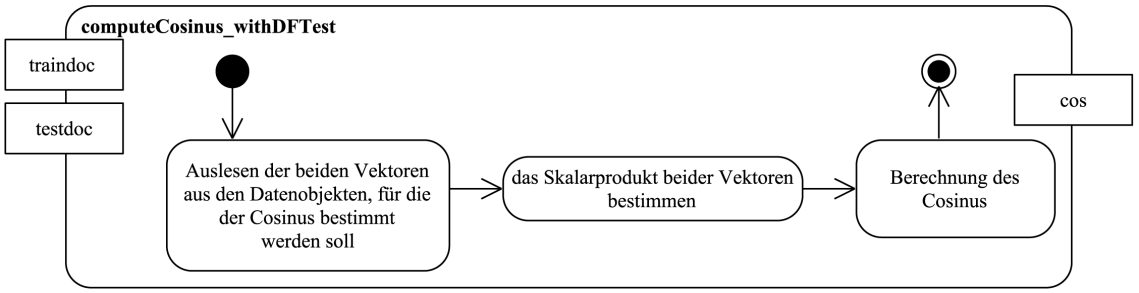


Abbildung 4.53: Ablauf der Operation `computeCosinus_withDFTest`

4.1.9.2.2 Klassifizieren mit Text-Suffix-Fragment-Features mit dem Decision-Tree-Algorithmus

Das Klassifizieren der Testdokumente durch den Decision-Tree-Algorithmus erfolgt, indem für jedes Dokument der Baum von der Wurzel traversiert wird, bis eine Klasse zugeordnet werden kann.[1] Das ist dann der Fall, wenn ein Blattknoten des Baumes erreicht ist. Zu sehen ist der Ablauf der Operation `categorizeWithDecTree` in Abbildung 4.54.

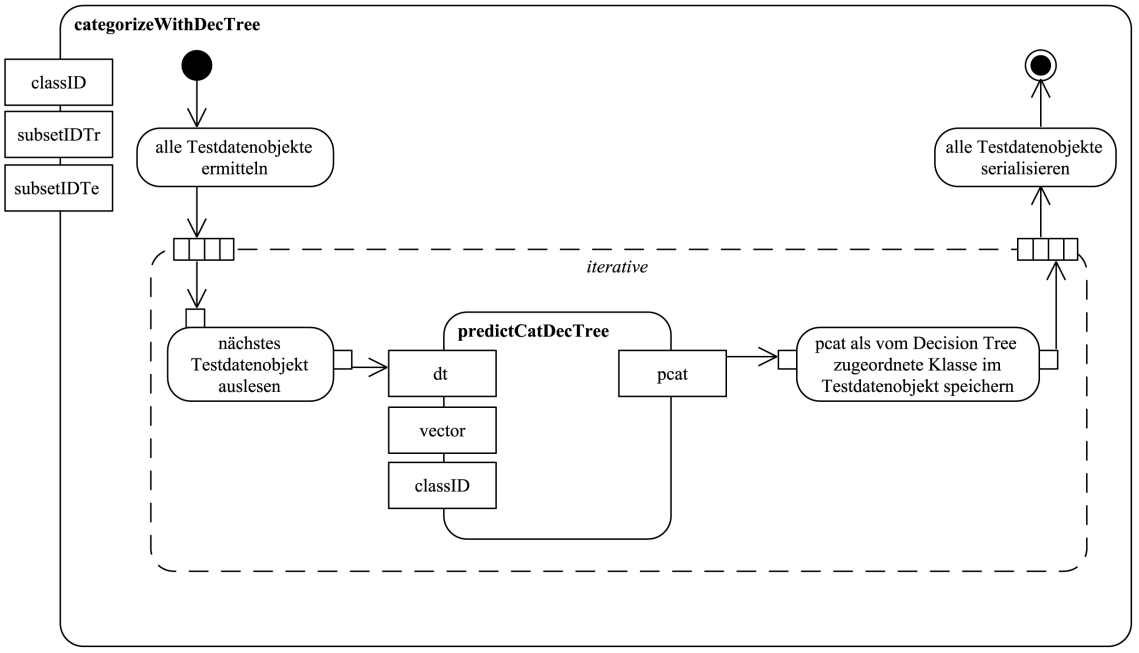


Abbildung 4.54: Ablauf der Operation `categorizeWithDecTree`

1 Auch hier gilt, dass in den Experimenten nur eine Klasse zugeordnet wird und daher auch nur dieser Fall implementiert wurde. Vorschläge für eine Klassifizierung, bei der mehrere Klassen einem Dokument zugewiesen werden können, finden sich in Kapitel 2.3.2.2.

Der Baum wurde mit Hilfe der Trainingsdokumente einer Teilmenge aufgebaut. Dabei steht jeder Knoten im Baum für ein TSF-Feature der Trainingsdokumente. Je nachdem, ob das TSF-Feature im Vektor des jeweiligen Testdokuments gesetzt oder nicht gesetzt ist, wird, vom aktuellen Knoten ausgehend, der nächste Knoten besucht.
Das Besuchen der Nachfolgeknoten erfolgt so lange, bis ein Blattknoten erreicht wird. Die dort angegebene Klasse der gewünschten Klassenfamilie ist die zuzuordnende Klasse für das Testdokument. Nachdem alle Testdokumente klassifiziert wurden, werden sie serialisiert, um evaluiert werden zu können.

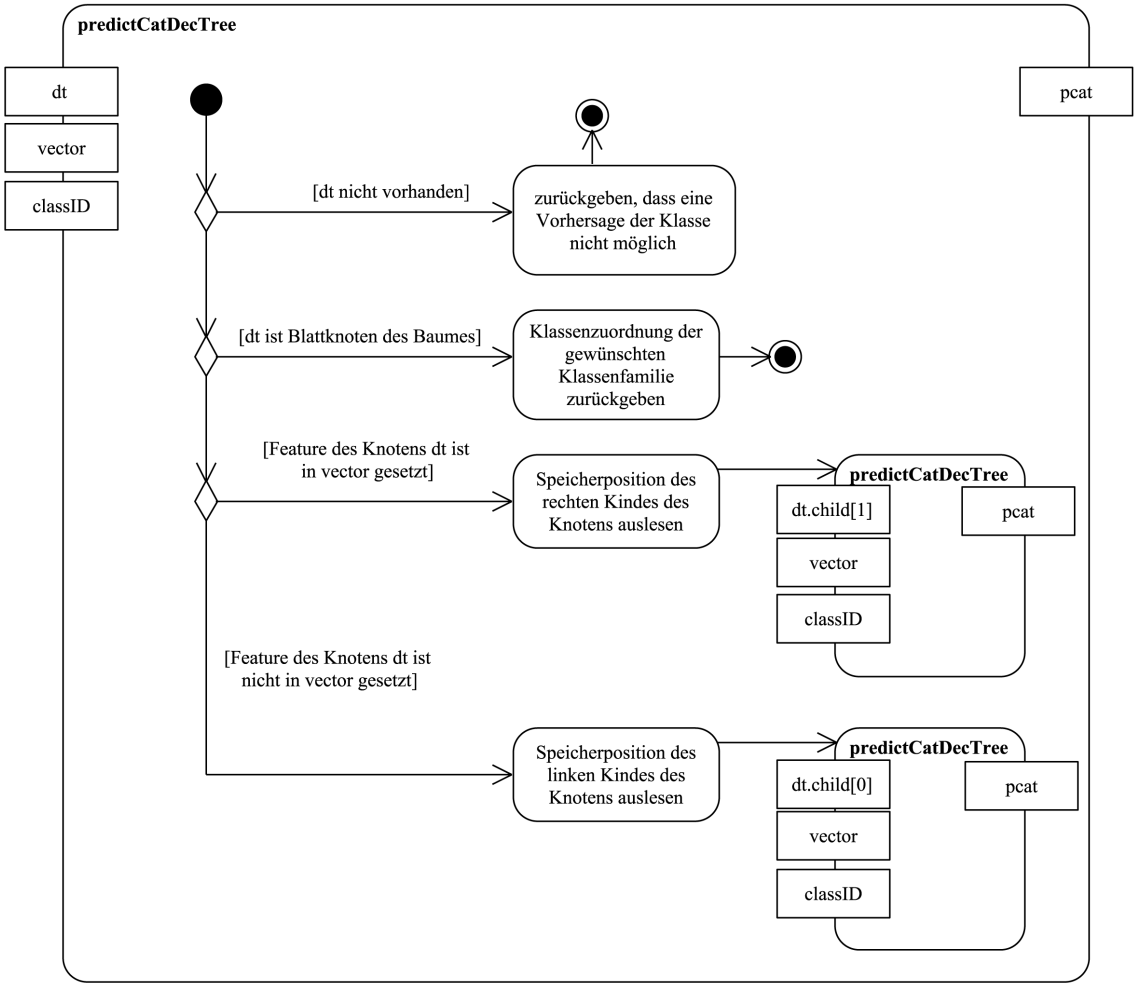


Abbildung 4.55: Ablauf der Operation `predictCatDecTree`

4.1.9.2.3 Klassifizieren mit Text-Suffix-Fragment-Features mit dem Naive-Bayes-Algorithmus

Das Klassifizieren mit dem Naive-Bayes-Algorithmus, siehe Kapitel 2.3.3.1, beruht auf einer Bewertung, die für jedes Testdokument für jede Klasse durchgeführt wird. Die Klasse mit der größten Bewertung wird letztendlich als die Klasse, zu der das Dokument gehört, betrachtet.[1]
Um die Berechnung durchführen zu können, müssen sowohl alle Testdatenobjekte als auch alle Klassen durchlaufen werden. In der Operation `computeNaiveBayesTest`, zu sehen in Abbildung 4.56 auf S. 288, wird in einer äußeren Schleife jedes Testdatenobjekt einzeln betrachtet.
Innerhalb dieser Schleife erfolgt eine Iteration über alle vorhandenen Klassen der betrachteten Klassenfamilie. Jede dieser Klassen wird eingelesen und ihre Dokumentanzahl bestimmt. Diese bezieht sich auf die Anzahl der der Klasse zugeordneten Trainingsdokumente, da die Anzahl der zugeordneten Testdokumente zu diesem Zeitpunkt nicht bekannt ist. Hier müssen zwei Fälle unterschieden werden:

1. Es existieren Trainingsdokumente mit dieser Klassenzuordnung.

2. Es existieren keine Trainingsdokumente mit dieser Klassenzuordnung.

Im zweiten Fall darf keine weitere Berechnung durchgeführt werden, da der letzte Teil der Formel 2.10, zu sehen auf S. 46, darauf beruht, die Anzahl der Dokumente der Klasse durch die Gesamtanzahl der Dokumente zu teilen und auf dieses Ergebnis den natürlichen Logarithmus anzuwenden. Ist aber die Anzahl der Dokumente der Klasse Null, so würde der natürliche Logarithmus von Null berechnet. Dieser ist nicht definiert. Das bedeutet in diesem Fall, dass mit der nächsten vorhandenen Klasse fortgefahren wird. Des Weiteren bedeutet das auch, dass kein Testdokument jemals einer Klasse ohne wenigstens ein zugehöriges Trainingsdokument zugeordnet werden kann.
Im ersten Fall erfolgt eine Berechnung der Wahrscheinlichkeit, dass das gerade betrachtete Testdokument der gerade betrachteten Klasse zugeordnet wird. Dafür wird der Vektor des Dokuments betrachtet und für jedes gesetzte TSF-Feature die Berechnung innerhalb der Summe in der Formel 2.10 auf S. 46 durchgeführt. Das Ergebnis der Berechnung wird jeweils auf die zuvor berechneten Werte aufaddiert, so dass nach dem Durchlaufen aller gesetzten TSF-Features die Summe aus der Formel

1 Bei der Verwendung des Naive-Bayes-Algorithmus wird ebenfalls in den Experimenten jedem Dokument nur eine Klasse zugeordnet. Vorschläge für die Zuordnung mehrerer Klassen zu einem Dokument finden sich in Kapitel 2.3.2.2.

komplett berechnet ist. Zusätzlich wird auf diese Summe die Anzahl der Dokumente der Klasse dividiert durch die Anzahl der Trainingsdokumente aufaddiert. Der Quotient wird in einem nachfolgenden Schritt berechnet. Die so berechnete Bewertung wird zum einen im Datenobjekt des Testdokuments an der Stelle der gerade betrachteten Klasse gespeichert und zum anderen wird geprüft, ob es sich um die bisher maximale Bewertung handelt. Ist letzteres der Fall, so wird die ID der Klasse als die Klasse mit der maximalen Bewertung für das Testdokument gespeichert.
Nachdem alle Klassen der ausgewählten Klassenfamilie für das Testdokument betrachtet wurden, steht die Klasse mit der maximalen Bewertung fest und wird als durch den Naive-Bayes-Algorithmus zugeordnete Klasse im Datenobjekt des Testdokuments gespeichert. Sind alle Testdokumente auf diese Art bearbeitet worden, können alle Testdatenobjekte serialisiert werden, um später immer wieder auf die Klassenzuordnungen zugreifen zu können.

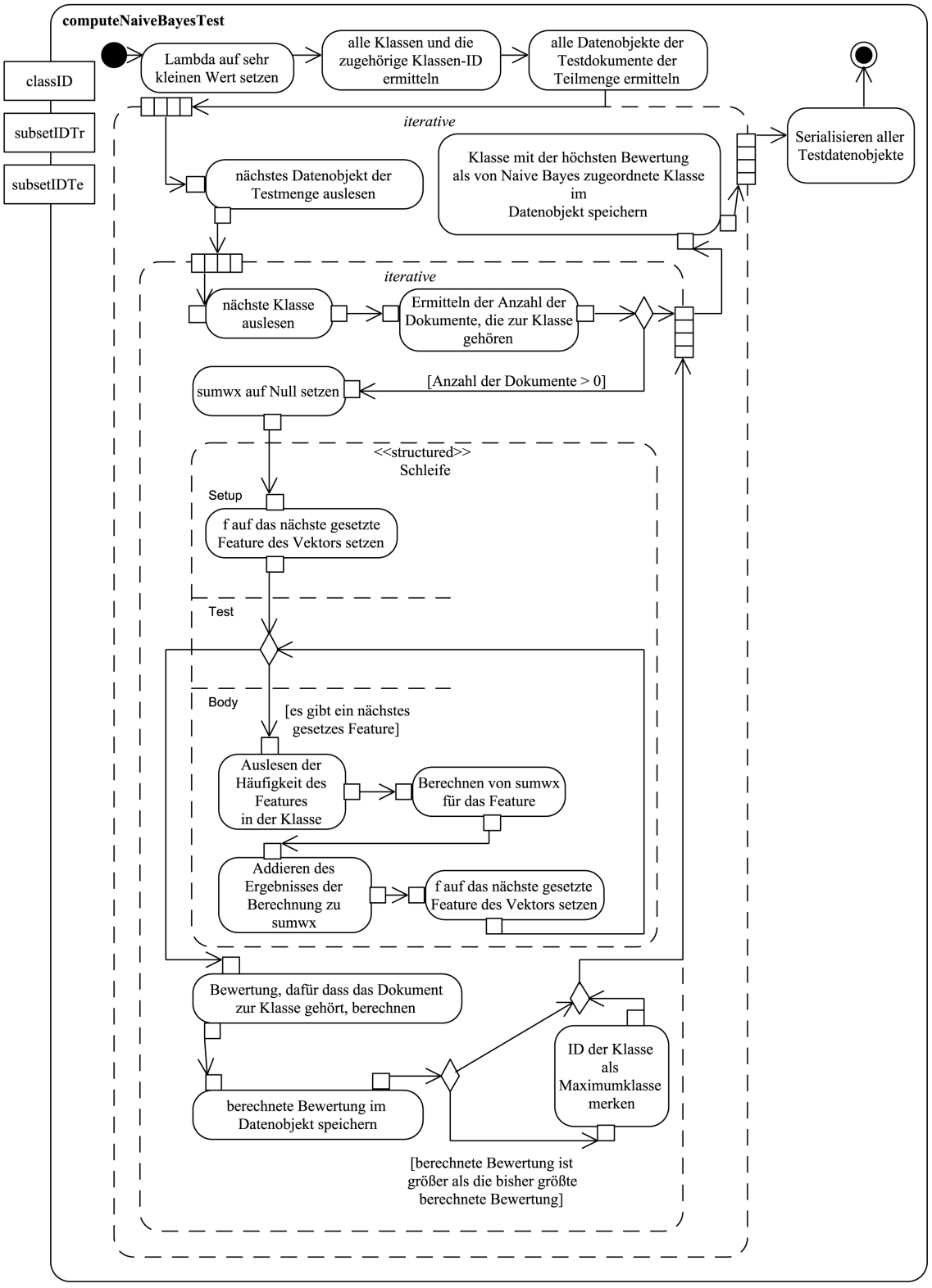


Abbildung 4.56: Ablauf der Operation `computeNaiveBayesTest`

4.1.9.2.4 Klassifizieren mit Text-Suffix-Fragment-Features mit dem Support-Vector-Machine-Algorithmus

Das Klassifizieren der Testdokumente mit dem Support-Vector-Machine-Algorithmus erfolgt mit der Software SVM^{light} von Joachims (2008a). Diese Software implementiert eine Support Vector Machine in der Programmiersprache C. Diese Software wurde ausgewählt, da sie über eine Implementierung verfügt, die nicht binär zwischen allen möglichen Klassen entscheidet, welche Klasse dem Dokument zugewiesen wird, sondern zwischen allen möglichen Klassen direkt entscheidet. Diese Implementierung namens SVM^{multiclass} wird in der vorliegenden Arbeit zum Klassifizieren der Testdaten verwendet.[1] Da diese Art des Entscheidens von den anderen implementierten Algorithmen ebenfalls durchgeführt wird, wurde diese Software ausgewählt. Der Ablauf des Klassifizierens mit SVM^{multiclass} ist wie bei den anderen Algorithmen:

1. Trainieren des Klassifizierers mit der gewünschten Trainingsteilmenge
2. Klassifizieren der gewünschten Testteilmenge mit dem erzeugten Klassifizierer

Im ersten Schritt wird mit Hilfe der erzeugten Vektordatei für den Support-Vector-Machine-Algorithmus und der Software SVM^{multiclass} eine SVM trainiert, die die Trainingsteilmenge mit einem möglichst kleinen Fehler klassifiziert. Dafür wird die Operation `svm_multiclass_learn` mit dem Parameter $C = 1,0$ aufgerufen, der die Vektordatei übergeben wird. Diese Operation erzeugt eine Ausgabedatei mit dem Modell der trainierten SVM. Der Parameter C legt laut Joachims (2008b) das Verhältnis zwischen dem zulässigen Trainingsfehler und der Breite des Randes fest. Dabei handelt es sich um den Parameter, der in Kapitel 2.3.3.4 auf S. 60 erläutert wird. Der Wert $1,0$ wird von Joachims (2008b) in seinem Beispiel vorgeschlagen und deshalb von der Verfasserin übernommen. Es wird ein linearer Kernel gewählt.[2]
Nachdem die SVM trainiert ist, können die Testdaten mit ihr klassifiziert werden. Dazu wird die Operation `svm_multiclass_classify` aufgerufen. Diese erhält die

1 Die Implementierung stammt ebenfalls von Joachims (2008b).
2 Die beiden Parameter werden ohne weitere Validierungen übernommen, da es in den Experimenten dieser Arbeit nur um den Vergleich der Klassifizierungsergebnisse zwischen dem Verfahren mit TSF-Features und dem wortbasierten Verfahren geht, nicht aber um ein qualitativ sehr gutes Klassifizierungsergebnis. Um sehr gute Klassifizierungsergebnisse mit einer Support-Vector-Machine erzielen zu können, sollten sowohl die Parameter, in diesem Fall das C, als auch die Entscheidung, welcher Kernel verwendet wird, durch Kreuzvalidierung, siehe S. 51 dieser Arbeit, ermittelt werden, vgl. Bennett u.a. (2000), S. 10. Da das Training einer SVM, insbesondere bei der gegebenen Menge an TSF-Features, sehr aufwändig sein kann, vgl. Noble (2006), S. 1567, wurde in der vorliegenden Arbeit auf die Festlegung der Parameter durch Kreuzvalidierung verzichtet.

Vektordatei mit den Feature-Vektoren für die SVM für die gewünschte Testteilmenge und die beim Trainieren der SVM erzeugte Modelldatei für die gewünschte Trainingsteilmenge. Als Ausgabe wird eine Datei erzeugt, die pro Zeile die zugeordnete Klasse für den Feature-Vektor - also das entsprechende Dokument - aus der übergebenen Vektordatei enthält.

4.1.10 Evaluation des Klassifizierens mit Text-Suffix-Fragment-Features

4.1.10.1 Evaluation durch Berechnung des Anteils der korrekt klassifizierten Dokumente an allen zu klassifizierenden Dokumenten

4.1.10.1.1 Evaluation für die Naive-Bayes-, Decision-Tree- und k-Nearest-Neighbour-Algorithmen

Werden die Testdokumente genau einer Klasse einer Klassenfamilie zugeordnet, so reicht es für die Evaluation aus, zu überprüfen, wie viele Testdokumente korrekt zugeordnet wurden. Dafür wird die Anzahl der korrekt zugeordneten Testdokumente bestimmt und ihr Anteil an der Gesamtanzahl der Testdokumente[1] berechnet. Der Ablauf bei der Ermittlung der Anzahl der korrekt klassifizierten Testdokumente unterscheidet sich leicht je nach verwendetem Algorithmus zum Klassifizieren.
Bevor die eigentliche Evaluierungsmethode angestoßen werden kann, müssen zunächst die gespeicherten Testdaten, die die Klassenzuordnungen enthalten, wieder eingelesen werden, so dass eine Betrachtung der den Testdokumenten zugeordneten Klassen möglich ist. Das erfolgt in der Operation `prepareEvaluationSubsetsTest`, zu sehen in Abbildung 4.57 auf S. 291.
Wurde mit dem k-Nearest-Neighbour-Algorithmus klassifiziert, so wird die Operation `evaluatekNNTestOneClass`, siehe Abbildung 4.58 auf S. 291, ausgeführt. Sie ermittelt für jedes Testdokument die durch kNN zugeordnete Klasse der betrachteten Klassenfamilie. Zudem erfolgt die Ermittlung der ursprünglich von Reuters zugeordneten Klasse oder Klassen.[2] Daraufhin wird überprüft, ob die durch den kNN zugewiesene Klasse in den durch Reuters zugewiesenen Klassen enthalten ist. Ist das der Fall, so wird die Anzahl der korrekt zugeordneten Testdokumente um Eins erhöht. Dann wird, genau wie im Fall, dass nicht die korrekte Klasse zugeordnet wurde, mit dem nächsten Testdokument der Teilmenge fortgefahren. Sind alle Testdokumente betrachtet worden, wird der Prozentsatz der korrekt klassifi-

1 Die genauen Parameter der einzelnen Experimente können in Kapitel 4.3.2 nachgelesen werden.
2 Die Testteilmengen sind, anders als die Trainingsteilmengen, nicht in allen Fällen auf eine Klasse der zugeordneten Klassen einer Klassenfamilie beschränkt, siehe auch Kapitel 4.3.2.

zierten Testdokumente berechnet, indem die ermittelte Anzahl auch hier durch die Gesamtanzahl an Testdokumenten geteilt wird.

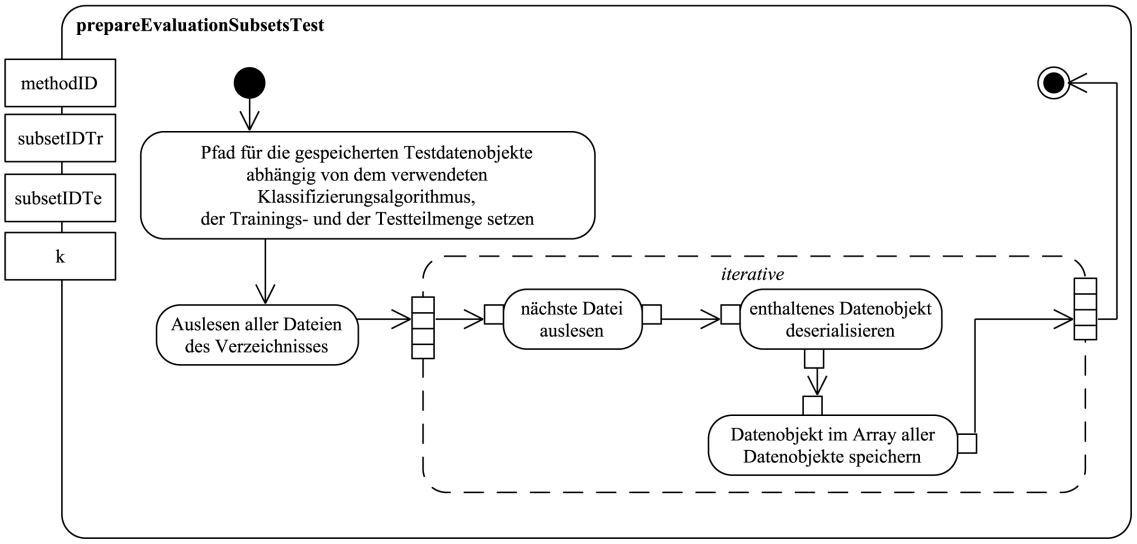


Abbildung 4.57: Ablauf der Operation `prepareEvaluationSubsetsTest`

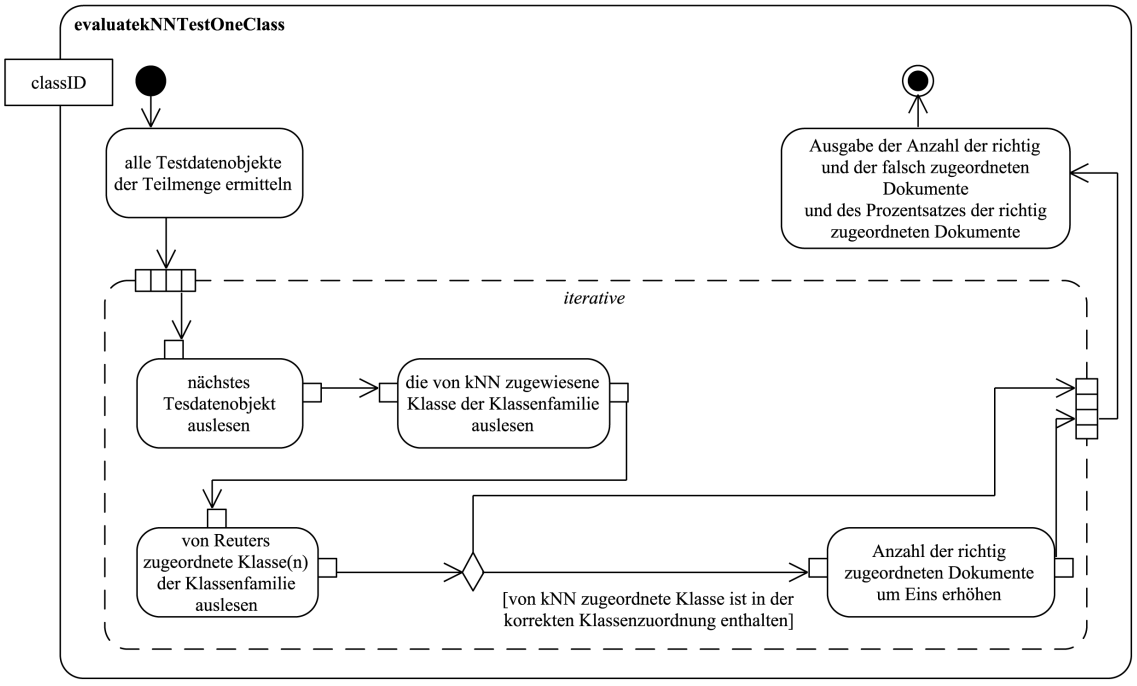


Abbildung 4.58: Ablauf der Operation `evaluatekNNTestOneClass`

Für die Evaluation der Klassifizierung durch den Decision Tree ist die Verarbeitung sehr ähnlich. Sie wird durch den Aufruf der Operation `evaluateDTTestOneClass`, zu sehen in Abbildung 4.59, angestoßen. Der Unterschied besteht hier darin, die durch den Decision Tree getroffene Klassenzuordnung des Testdokuments auszulesen und mit der Klassenzuordnung von Reuters zu vergleichen. Die Berechnung des Anteils der korrekt klassifizierten Testdokumente erfolgt auf die gleiche Art und Weise, wie beim k-Nearest-Neighbour-Algorithmus.

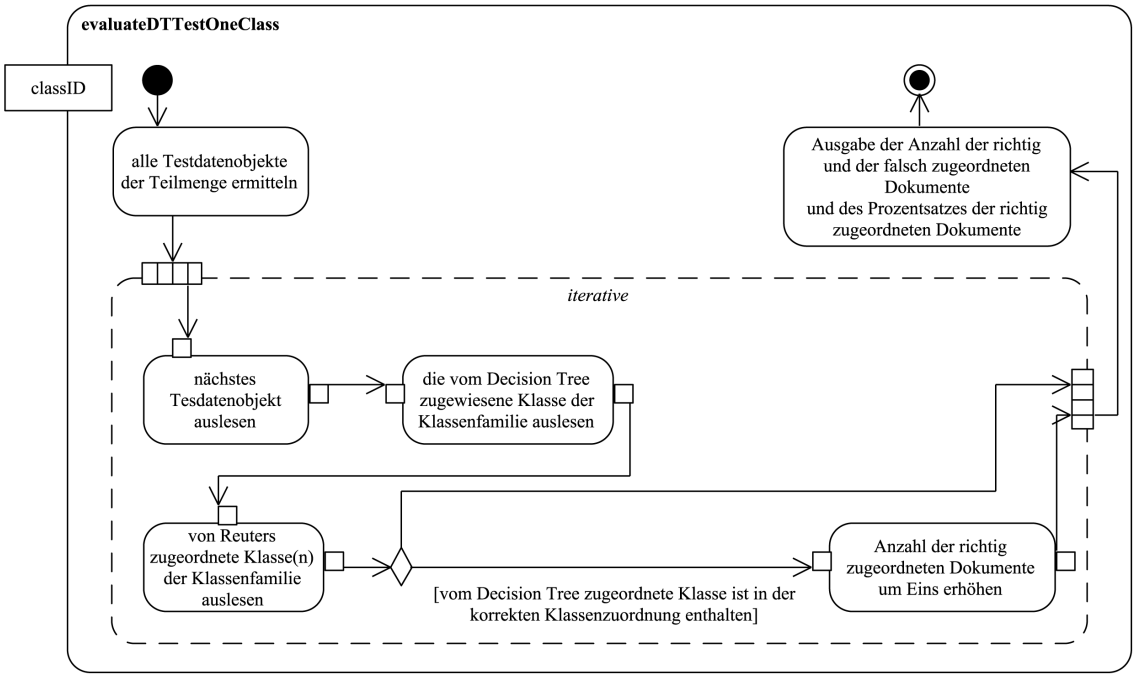


Abbildung 4.59: Ablauf der Operation `evaluateDTTestOneClass`

Auch die Evaluation der Klassenzuordnung durch den Naive-Bayes-Algorithmus läuft nach dem gleichen Schema ab, siehe Abbildung 4.60 auf S. 293. Hier wird die durch den Naive-Bayes-Klassifizierer zugeordnete Klasse eines jeden Testdokuments ausgelesen und mit der durch Reuters getroffenen Zuordnung verglichen. Die Berechnung des prozentualen Anteils der korrekt klassifizierten Testdokumente erfolgt ebenfalls wie zuvor beschrieben.

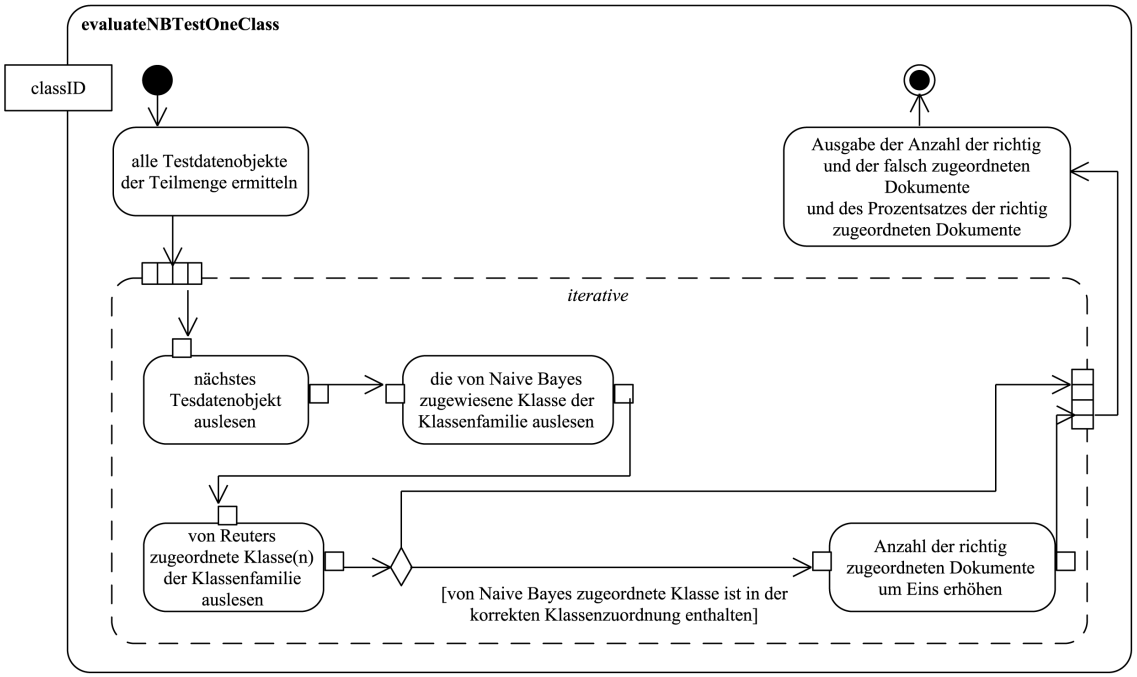


Abbildung 4.60: Ablauf der Operation `evaluateNBTestOneClass`

4.1.10.1.2 Evaluation für den Support-Vector-Machine-Algorithmus

Bevor die Evaluation, wie für die anderen Algorithmen beschrieben, für die SVM durchgeführt werden kann, muss zunächst die von der SVM erzeugte Ausgabedatei in die Datenobjekte umgewandelt werden, die durch die entsprechenden Java-Operationen verarbeitet werden können. Dazu wird die Operation `prepareEvaluationSubsetsTestSVMMulticlass` aufgerufen. Diese liest sowohl die bei der Erzeugung der SVM-Vektordatei erzeugte Mappingdatei ein als auch die durch die SVM erzeugte Ausgabedatei. Man benötigt beide Dateien, da in der Ausgabedatei lediglich die durch die SVM zugeordnete Klasse für das Dokument steht, jedoch nicht die ID des Dokuments. Diese erhält man durch die Verbindung zwischen Zeilennummer und `overallID` aus der Mappingdatei.

Man speichert sowohl die Kombination aus `overallID` und Zeilennummer in einer Datenstruktur namens `mapping` und die Kombination aus Zeilennummer und zugeordneter Klassen-ID in einer Datenstruktur namens `mappingOutput`. Anschließend wird über `mapping` iteriert und jeweils die nächste `overallID` ausgelesen. Für diese wird ein Datenobjekt erzeugt, das dem Datenobjekt für die Evaluation der anderen Algorithmen entspricht. In diesem Datenobjekt wird die jeweilige von Reuters zugeordnete Klasse gespeichert und auch die von der Support-Vector-Machine zu-

gewiesene Klasse. Das erfolgt für alle Dokumente der Testteilmenge. Abschließend erfolgt das Speichern aller Datenobjekte der Testteilmenge. Eine Darstellung der Operation befindet sich in Abbildung 4.61.

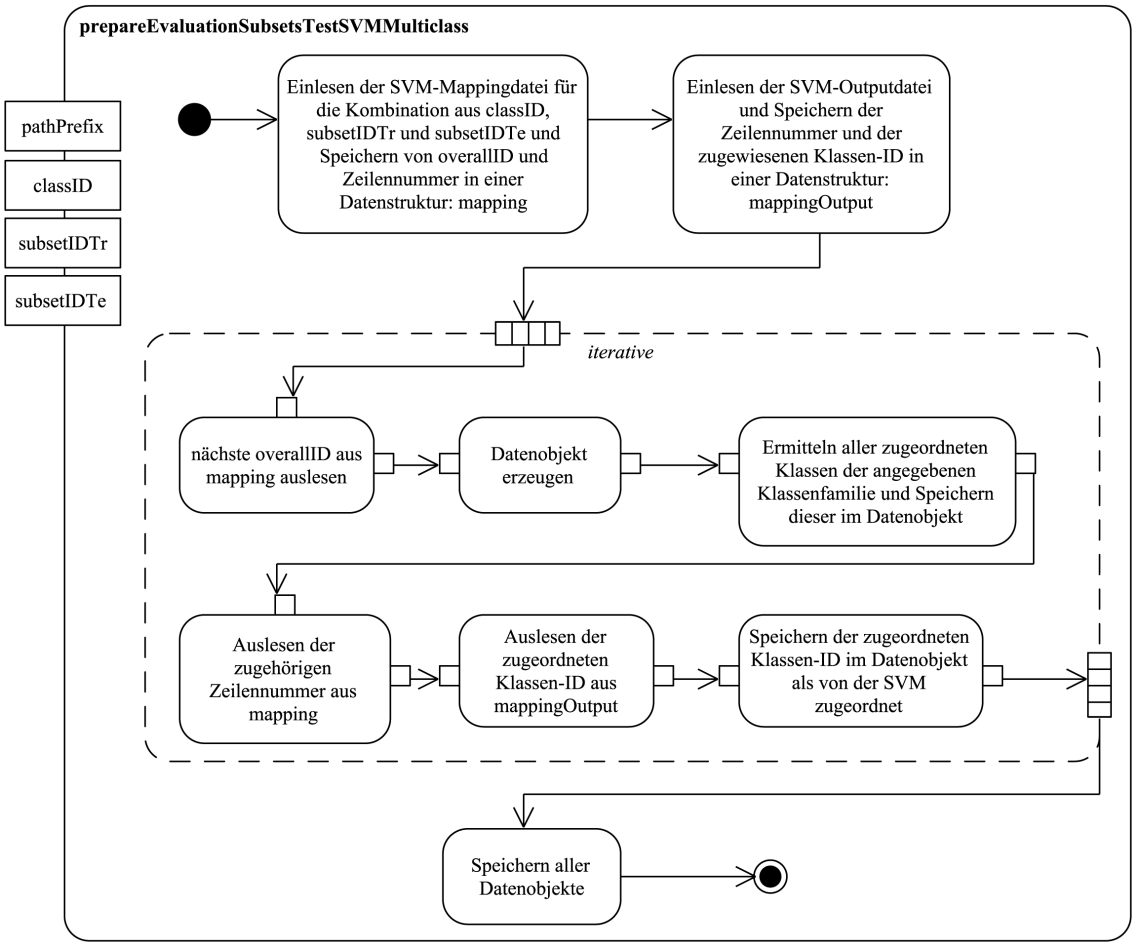


Abbildung 4.61: Ablauf der Operation `prepareEvaluationSubsetsTestSVMMulticlass`

Das Speichern ist nötig, damit die in Kapitel 4.1.10.1.1 beschriebene Operation `prepareEvaluationSubsetsTest` und anschließend die Operation `evaluateWithMethodSubsetsTest` ausgeführt werden können und die gleiche Verarbeitung durchgeführt wird wie für die anderen Algorithmen. Der einzige Unterschied besteht darin, dass durch die Operation `evaluateWithMethodSubsetsTest` nun die Operation `evaluateSVMTestOneClass` aufgerufen wird.

Diese unterscheidet sich jedoch nur in einem einzigen Punkt von den Operationen der anderen Algorithmen: Hier wird die von der SVM zugeordnete Klasse mit der von Reuters zugeordneten Klasse verglichen und entsprechend der Anteil der richtig zugeordneten Dokumente, gemessen an allen zu klassifizierenden Dokumenten, erhöht. Zu sehen ist die Operation in Abbildung 4.62.

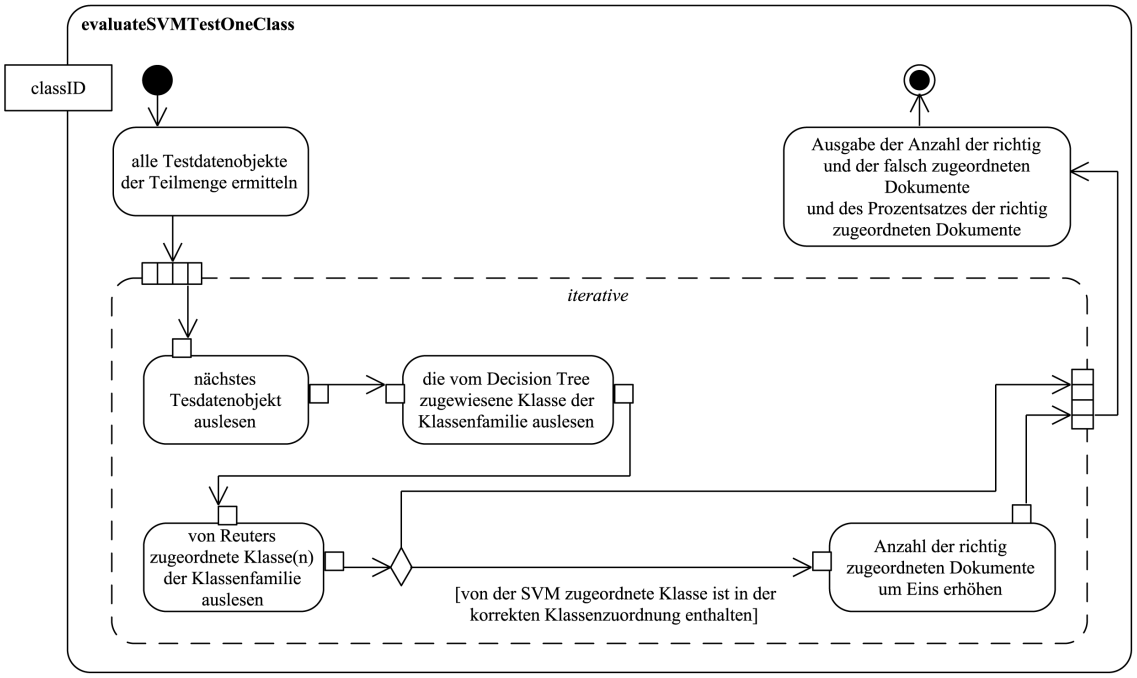


Abbildung 4.62: Ablauf der Operation `evaluateSVMTestOneClass`

4.1.10.2 Evaluation durch Verwendung des F-Maßes

Neben der Berechnung des Anteils der korrekt zugeordneten Dokumente wird auch das F_1-Maß benutzt, um die Qualität der Klassifizierung zu bewerten. Wie der Name bereits sagt, wird der Parameter $\beta = 1$ gesetzt, also werden Genauigkeit und Trefferquote gleichgewichtet. Zusätzlich werden jeweils das microaveraging und das macroaveraging benutzt, um die Qualität der Klassifizierung über alle Klassen festzustellen.
Die Vorbereitung der Evaluierung läuft genauso ab, wie im Kapitel 4.1.10.1 beschrieben. Danach erfolgt ebenfalls der Aufruf der Operation `evaluateWithMethodSubsetsTest`, die aber nun die Operation für die Berechnung des F_1-Maßes aufruft.
Dabei handelt es sich um die Operation `evaluateTestOneClassF1`, zu sehen in Abbildung 4.63 auf S. 297. Sie beginnt damit, alle Klassen, die den Testdokumenten zugewiesen werden können, die also in den Trainingsdaten vorkommen, in einer Da-

tenstruktur zu speichern. Zusätzlich wird eine Datenstruktur namens `allClasses` erzeugt, die pro Klasse ein Objekt enthält, das die für die Berechnung von F_1 benötigten Daten speichern kann. Dabei handelt es sich um die Möglichkeit zum Speichern von TP, FP, FN und F_1 pro Klasse. Zusätzlich kann in dieser Datenstruktur pro Klasse hinterlegt werden, welche Dokumente durch den zu evaluierenden Klassifizierer der Klasse zugeordnet wurden und welche Klassen laut der Reuters-Klassifikation zur Klasse gehören. Diese zuletzt genannten Informationen sind wichtig, um die TP, FP und FN richtig zählen zu können.
Anschließend wird über die Datenobjekte der Testdaten iteriert. Für jedes Datenobjekt wird ermittelt, welcher Klasse es durch den Klassifizierer zugeordnet wurde und welches die korrekte Klasse ist. Diese Informationen werden in der zuvor genannten Datenstruktur `allClasses` gespeichert. Nachdem alle Informationen aus den klassifizierten Testdaten auf diese Art verarbeitet wurden, wird über `allClasses` iteriert, um die TP, FP und FN pro Klasse zu ermitteln. Dafür wird geprüft, ob die der Klasse durch den Klassifizierer zugewiesenen Dokumente in den korrekten Dokumenten der Klasse enthalten sind. Ist das der Fall, werden die TP um eins erhöht, ansonsten die FP. Zudem wird auch der umgekehrte Fall geprüft, also dass Dokumente nicht zugeordnet wurden, obwohl sie hätten zugeordnet werden müssen, das sind die FN pro Klasse.
Sind diese Anzahlen für jede einzelne Klasse ermittelt worden, so wird nochmals über die Klassen iteriert und nun der F_1-Wert pro Klasse berechnet und der Gesamtsumme der F_1-Werte über alle Klassen hinzuaddiert. Außerdem werden die Gesamtanzahlen TP, FP und FN um die entsprechende Anzahl in der aktuellen Klasse erhöht. Nachdem das für alle Klassen durchgeführt wurde, wird der macroaveraged F_1-Wert durch das Teilen der Summe der F_1-Werte durch die Anzahl der zuordenbaren Klassen berechnet. Der microaveraged F_1-Wert wird anhand der aufsummierten Werte für TP, FP und FN über alle Klassen berechnet. Abschließend werden beide Ergebnisse ausgegeben.

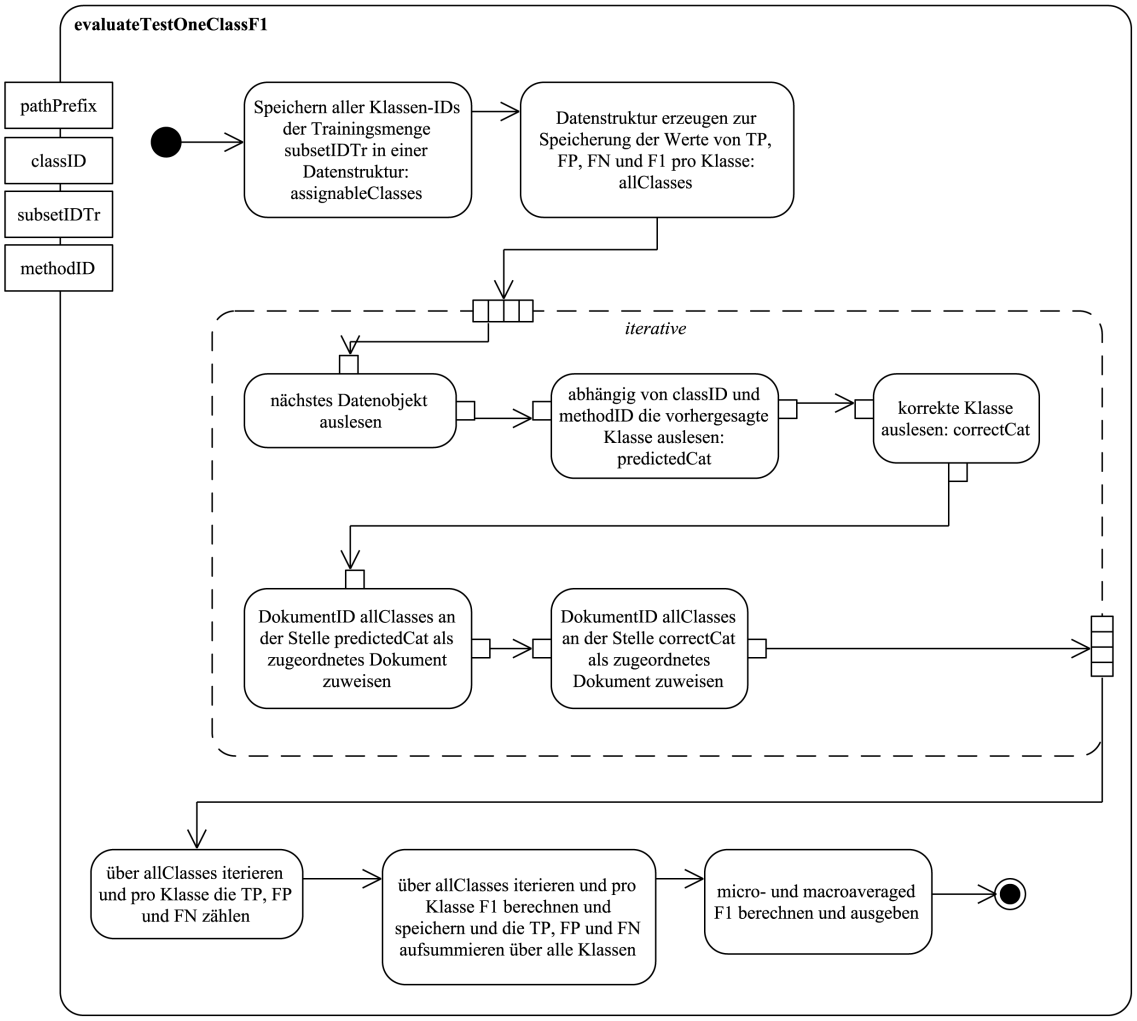


Abbildung 4.63: Ablauf der Operation `evaluateTestOneClassF1`

4.2 Implementierung des Klassifizierens mit Einzelwort-Features

4.2.1 Einleitung in die Implementierung des Klassifizierens mit Einzelwort-Features

Um vergleichbare Ergebnisse zu erzielen, müssen die gleichen Verarbeitungsschritte mit den von D.D. Lewis zur Verfügung gestellten Feature-Vektoren durchgeführt werden, die bei den auf TSF-Features basierenden Feature-Vektoren durchgeführt wurden. Hier ist jedoch der Vorteil gegeben, dass die Feature-Vektoren bereits vorliegen, so wie sie von D.D. Lewis in seinen Experimenten benutzt wurden. Das bedeutet, sie müssen nicht erzeugt werden, sondern lediglich in eine Form gebracht

werden, die den auf TSF-Features basierenden Feature-Vektoren entspricht, um sie dann in den Algorithmen zum Klassifizieren verwenden zu können.
Insgesamt bedeutet das also, dass die ersten Schritte von der Erzeugung des RCV1-v2 bis einschließlich des Herausfilterns von Dokumenten mit nur einer Klassenzugehörigkeit hier entfallen und nur eine Beschreibung der Vorbereitung der Trainings- und Testdaten, sowie des Klassifizierens und der Evaluation erfolgt.

4.2.2 Trainingsdaten vorbereiten für das Klassifizieren mit Einzelwort-Features

4.2.2.1 Trainings-Teilmengen erstellen für das Klassifizieren mit Einzelwort-Features

Das Erstellen der Teilmengen beschränkt sich im Fall der Lewis-Feature-Vektoren darauf, die Feature-Vektoren derjenigen Dokumente herauszufiltern, die in den entsprechenden Teilmengen bei der Verwendung der auf TSF-Features basierenden Feature-Vektoren vorhanden sind. Das erfolgt parallel zur Erstellung der Lewis-Feature-Vektoren, siehe nächstes Unterkapitel.

4.2.2.2 Erzeugen der Lewis-Feature-Vektoren für die Trainingsdaten

D.D. Lewis stellt auf einer Webseite[1] für alle Dokumente eine Datei mit den entsprechenden Trainingsvektoren und vier Dateien mit den entsprechenden Testvektoren zur Verfügung. Diese Dateien können heruntergeladen und verwendet werden. Sie sind nach folgendem Schema aufgebaut[2]:

- Jede Zeile in jeder Datei beinhaltet den Lewis-Feature-Vektor genau eines der Dokumente des RCV1-v2.
- Jeder Lewis-Feature-Vektor beginnt mit der Reuters-ID des entsprechenden Dokuments.
- Nach dieser ID befinden sich zwei Leerzeichen.

1 Vgl. Lewis (2004).
2 Vgl. Lewis (2004), Appendix 13.

- Darauf folgen IDs und Gewicht aller Features, die einen Wert im Dokument haben. Dabei sind Feature-ID und Gewicht durch einen Doppelpunkt getrennt und die einzelnen Features durch ein Leerzeichen.[1]

Ein Ausschnitt aus einer der Dateien als Abbildung 4.64 verdeutlicht den Dateiaufbau.

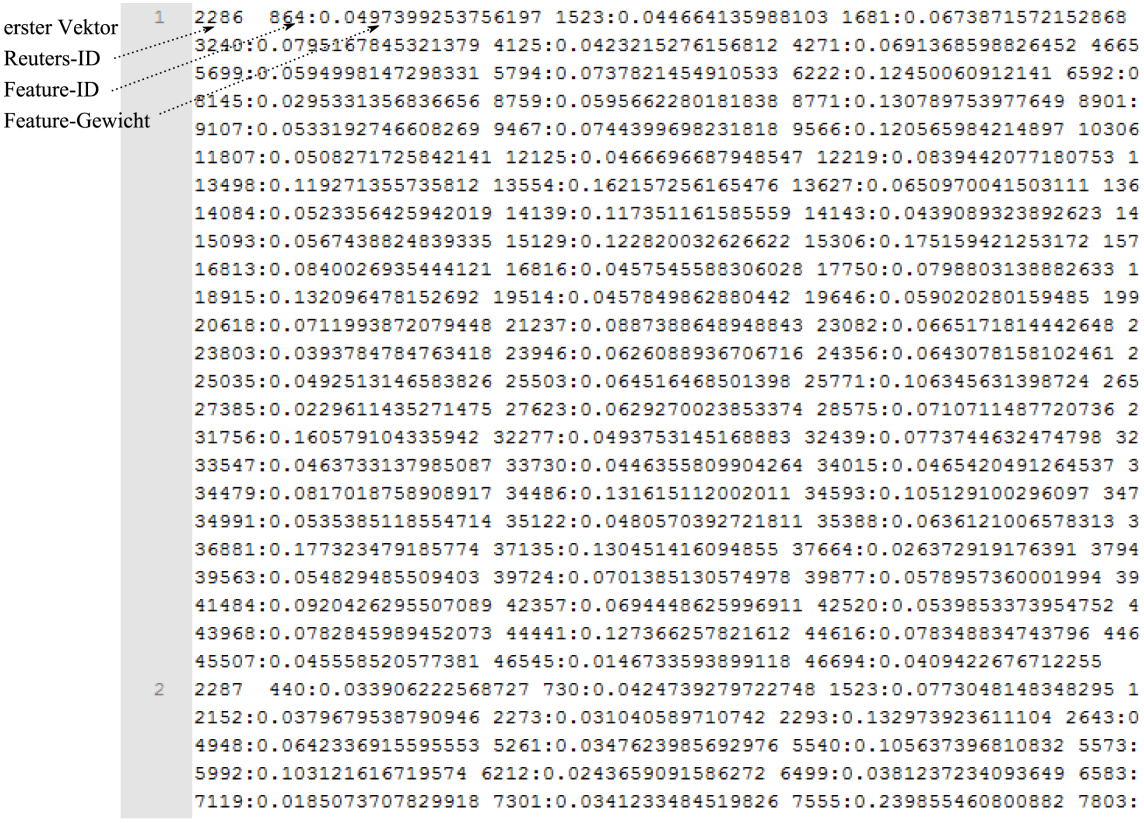


Abbildung 4.64: Dateiausschnitt mit Lewis-Feature-Vektoren in ursprünglicher Form

Für die Algorithmen zum Klassifizieren, wie sie in dieser Arbeit verwendet werden, ist es nur entscheidend, ob ein bestimmtes Feature in einem Dokument vorhanden ist

1 Genauso wie im Fall der TSF-Features wurden ebenfalls nur die Inhalte des <headline>-Tags und die <text>-Elemente der entsprechenden XML-Dateien verwendet, um die Lewis-Feature-Vektoren zu erzeugen. Diese Textkombination wurde in Kleinschreibung ohne Satzzeichen und Stoppworte verwendet. Zusätzlich fand ein so genanntes Stemming, damit bezeichnet man das algorithmische Zurückführen der Wörter auf eine Art „Grundform“, siehe Fußnote auf S. 10 für eine Erläuterung, der Wörter statt. Die so erhaltenen Features wurden gewichtet. Die Gewichtung der Features wird mit Hilfe einer Art „...$TF \times idf$...“, Lewis u.a. (2004), S. 387, erzeugt. TF steht dabei für *term frequency* und *idf* steht für *inverse document frequency*, also *inverse Dokumenthäufigkeit*. Für weitere Erläuterungen siehe Kapitel 2.1.2.5 auf S. 27. Zusätzlich werden noch Features selektiert und die Länge der entstandenen Vektoren normalisiert. Für eine genauere Beschreibung vgl. Lewis u.a. (2004), S. 386-388.

oder nicht. Der Vektor des entsprechenden Dokuments muss lediglich diese Information abbilden. Das bedeutet, dass die von Lewis zur Verfügung gestellten Vektoren vereinfacht werden können, um sie innerhalb der vorgestellten Algorithmen zum Klassifizieren verwenden zu können. Aus diesem Grund erfolgt ein erstes Parsen der Trainings-Vektordateien durch ein Python-Programm[1] mit dem Ziel, den Aufbau der Vektordateien wie folgt zu ändern:

- Jede Zeile einer Datei bildet weiterhin einen Lewis-Feature-Vektor ab, jedoch ändert sich der Aufbau.
- Zu Beginn steht immer noch die von Reuters vergebene ID des entsprechenden Dokuments.
- Darauf folgt ein Tab.
- Anschließend folgen, mit einem Leerzeichen voneinander getrennt, alle Feature-IDs aus der Originaldatei, jedoch ohne die dazugehörenden Gewichte, da sie hier in diesem Kontext keine Rolle spielen.

Dieser neue Aufbau ist in Abbildung 4.65 auf S. 301 zu sehen. Auch der geänderte Aufbau der ursprünglichen Vektordateien ist noch nicht die endgültige Form der Lewis-Feature-Vektoren, die für die hier verwendeten Algorithmen zum Klassifizieren benötigt wird. Um diese endgültige Form zu erstellen, werden zunächst die IDs der Dokumente ermittelt, die zu der entsprechenden Teilmenge gehören. Dafür werden die Suffix Arrays der Teilmenge eingelesen und ihre IDs - die von Reuters vergebene und die systemweit eindeutig von der Verfasserin vergebene ID, also die `overallID` - jeweils als Paar zusammengehörend in einer Datenstruktur gespeichert.
In einem nächsten Schritt werden die zuvor durch das Python-Programm erstellten Dateien zeilenweise - also Lewis-Feature-Vektor für Lewis-Feature-Vektor - eingelesen. Ist die entsprechende ID des Dokuments, zu dem der gerade eingelesene Lewis-Feature-Vektor gehört, in der Teilmenge vorhanden, so werden alle im Lewis-Feature-Vektor vorhandenen Feature-IDs in einer Datenstruktur angelegt und zusammen mit der eindeutigen ID gespeichert. Diese Form entspricht der Form der Feature-Vektoren, die aus den Suffix Arrays erstellt wurden.

1 Bei Python handelt es sich um eine Programmiersprache, vgl. Python Software Foundation (2010).

```
1  2286  864 1523 1681 2293 2845 2867 3240 4125 4271 4665 5216
   5573 5699 5794 6222 6592 7227 7975 8145 8759 8771 8901 8927
   8940 9107 9467 9566 10306 10326 11485 11807 12125 12219 12718
   12821 13321 13498 13554 13627 13658 13785 13861 14084 14139
   14143 14338 14716 14844 15093 15129 15306 15722 16499 16789
   16813 16816 17750 18120 18150 18269 18915 19514 19646 19946
   20116 20605 20618 21237 23082 23312 23578 23628 23803 23946
   24356 24438 24479 25035 25503 25771 26535 26805 26854 27385
   27623 28575 29781 31243 31594 31756 32277 32439 32674 33035
   33170 33547 33730 34015 34057 34204 34479 34486 34593 34763
   34910 34985 34991 35122 35388 35499 35945 36500 36881 37135
   37664 37940 39033 39418 39563 39724 39877 39913 39947 41225
   41484 42357 42520 428412 43325 43968 44441 44616 44654 44957
   45350 45507 46545 46694
2  2287  440 730 1523 1893 1897 2120 2152 2273 2293 2643 2963
   4125 4948 5261 5540 5573 5699 5990 5992 6212 6499 6583 6634
   6932 7119 7301 7555 7803 8145 8580 8605 8609 8646 8885 8901
   8915 8952 9055 9106 9265 9817 9946 9999 10293 10317 10709
   10711 10773 10953 11436 11502 12158 12219 12718 12764 12821
   12889 13421 13549 13573 13620 13778 14384 14823 15093 15456
```

Abbildung 4.65: Dateiausschnitt mit Lewis-Feature-Vektoren in umgewandelter Form

4.2.2.3 Erzeugen der Einzelwort-Feature-Datenobjekte für die Trainingsdaten

4.2.2.3.1 Erzeugen der Einzelwort-Feature-Datenobjekte für die Trainingsdaten für die Naive-Bayes-, Decision-Tree- und k-Nearest-Neighbour-Algorithmen

Aus den Vektoren werden auf die gleiche Weise Datenobjekte erzeugt wie bei der Verarbeitung mit den Suffix Arrays. Dafür wird zunächst ermittelt, welche Dokumente zu der entsprechenden Teilmenge gehören[1]. Der weitere Ablauf der Verarbeitung entspricht genau dem, der im Hinblick auf die Erzeugung der Datenobjekte bei der Verarbeitung mit Suffix Arrays beschrieben wurde, siehe S. 258 ff. Die einzigen Unterschiede bestehen in der festgelegten Anzahl an Features[2] und den anderen

1 Dadurch, dass die Dokumente selbst oder eine aufbereitete Form im Fall der Lewis-Feature-Vektoren nicht benötigt werden, da nur auf den Lewis-Feature-Vektoren gearbeitet wird, und diese gemeinsam in einer Datei gespeichert werden, müssen die IDs der einzelnen Dokumente mit Hilfe der Suffix Arrays der entsprechenden Teilmenge ermittelt werden. Das bedeutet, die entsprechenden IDs werden ausgelesen und in einer Datenstruktur gespeichert.

2 Im Fall der Lewis-Feature-Vektoren sind das 47.219 Features, vgl. Lewis u.a. (2004), S. 387. Im Fall der Suffix Arrays kann die Anzahl je nach betrachteter Klassenfamilie schwanken.

Verzeichnisnamen zum Speichern oder Auslesen der entsprechenden serialisierten Objekte.

4.2.2.3.2 Erzeugen der Einzelwort-Feature-Datenobjekte für die Trainingsdaten für den Support-Vector-Machine-Algorithmus

Da das Klassifizieren durch die Support-Vector-Machine mit einer Software durchgeführt wird, die nicht selbst implementiert wurde[1], unterscheidet sich das Erzeugen der Datenobjekte für die SVM von dem für die anderen Algorithmen.

Für die SVM müssen die Lewis-Feature-Vektoren der Trainingsdaten in das Eingabeformat der Software umgewandelt werden. Dazu wird die Operation `prepareCategorizationSubsetsSVMMulticlassLewis` aufgerufen. Sie unterscheidet sich nur insofern von der Operation für die TSF-Features als dass jetzt mit den erzeugten wortbasierten Lewis-Feature-Vektoren gearbeitet wird und diese in die gleiche Form gebracht werden, die als Eingabe für die SVM nötig ist. Auch die entsprechende Mappingdatei wird erzeugt.

4.2.3 Testdaten vorbereiten für das Klassifizieren mit Einzelwort-Features

4.2.3.1 Test-Teilmengen erstellen für das Klassifizieren mit Einzelwort-Features

Für die Testdaten gilt die gleiche Aussage wie bei den Trainingsdaten: Auch hier müssen lediglich die Lewis-Feature-Vektoren aus den von D.D. Lewis zur Verfügung gestellten Vektordateien herausgefiltert werden, die zur entsprechenden Teilmenge gehören. Das Verfahren entspricht dem im vorangegangenen Unterkapitel beschriebenen.

4.2.3.2 Erzeugen der Lewis-Feature-Vektoren für die Testdaten

Auch die Erzeugung der Lewis-Feature-Vektoren in der Form, dass sie in den hier verwendeten Algorithmen zum Klassifizieren benutzt werden können, erfolgt für die Testdaten nach dem gleichen Verfahren, wie für die Trainingsvektoren beschrieben.

1 Diese Software wird in Kapitel 4.1.9.2.4 auf S. 289 dieser Arbeit beschrieben.

4.2.3.3 Erzeugen der Einzelwort-Feature-Datenobjekte für die Testdaten

4.2.3.3.1 Erzeugen der Einzelwort-Feature-Datenobjekte für die Testdaten für den Naive-Bayes-, Decision-Tree- und k-Nearest-Neighbour-Algorithmus

Ebenso verhält es sich bei der Erzeugung der Datenobjekte. Auch sie werden nach dem gleichen Verfahren erzeugt, wie die Datenobjekte bei den TSF-Features-basierten Feature-Vektoren. Der einzige Unterschied besteht in den Speicherorten der entsprechenden serialisierten Objekte.

4.2.3.3.2 Erzeugen der Einzelwort-Feature-Datenobjekte für die Testdaten für den Support-Vector-Machine-Algorithmus

Da das Klassifizieren durch die Support-Vector-Machine mit einer Software durchgeführt wird, die nicht selbst implementiert wurde[1], unterscheidet sich das Erzeugen der Datenobjekte für die SVM von dem für die anderen Algorithmen.
Genau wie bei den Trainingsdaten wird ebenfalls die Operation `prepareCategorizationSubsetsSVMMulticlassLewis` aufgerufen. Sie führt genau die gleichen Schritte aus wie für die Trainingsdaten, arbeitet jedoch auf den Testdaten und erzeugt für diese die entsprechenden Vektordateien und Mappingdateien für die SVM.

4.2.4 Klassifizieren mit Einzelwort-Features

4.2.4.1 Vorbereiten des Klassifizierens mit Einzelwort-Features

4.2.4.1.1 Einlesen der Einzelwort-Feature-Datenobjekte der gewünschten Trainingsteilmenge

Das Verfahren zum Einlesen der Datenobjekte der gewünschten Trainingsteilmenge unterscheidet sich für die Lewis-Feature-Vektoren nicht von dem, das in Kapitel 4.1.9.1.1 im Hinblick auf die TSF-Feature-Vektoren vorgestellt wurde. Die einzigen Unterschiede beziehen sich auf die jeweiligen Speicherorte der entsprechenden Dateien und das Festlegen der Featureanzahl auf die vorhandene Anzahl, anstatt diese auszulesen.
Beim Klassifizieren mit der SVM entfällt das Einlesen der Datenobjekte, da das Klassifizieren durch eine Software durchgeführt wird. Die benötigten Dateien für die Eingabe wurden bereits in früheren Schritten erstellt.

1 Diese Software wird in Kapitel 4.1.9.2.4 auf S. 289 dieser Arbeit beschrieben

4.2.4.1.2 Einlesen der Einzelwort-Feature-Datenobjekte der gewünschten Testteilmenge

Auch das Einlesen der Datenobjekte der gewünschten Testteilmenge erfolgt wie für die TSF-Features-basierten Feature-Vektoren beschrieben. Die dafür verwendete Java-Operation unterscheidet sich lediglich durch den Verweis auf den entsprechenden Verzeichnispfad.

Auch das Einlesen der Datenobjekte für die Testteilmenge entfällt für die SVM aus den gleichen Gründen, wie beim Einlesen der Trainingsteilmenge erläutert.

4.2.4.2 Durchführen des Klassifizierens mit Einzelwort-Features

4.2.4.2.1 Klassifizieren mit Einzelwort-Features mit dem k-Nearest-Neighbour-Algorithmus

Das Klassifizieren mit dem k-Nearest-Neighbour-Algorithmus läuft für die Lewis-Feature-Vektoren genauso ab wie für die Feature-Vektoren auf Basis der TSF-Features. Auch hier beziehen sich die Unterschiede nur auf die verschiedenen Verzeichnispfade, die die entsprechenden Datenobjekte enthalten oder in die die entsprechenden Datenobjekte gespeichert werden.

4.2.4.2.2 Klassifizieren mit Einzelwort-Features mit dem Decision-Tree-Algorithmus

Für das Klassifizieren mit einem Decision Tree gilt ebenfalls, dass es genauso abläuft, wie im Kapitel 4.1.9.2.2 beschrieben. Es werden lediglich andere Verzeichnispfade zum Speichern oder Laden der entsprechenden Datenobjekte verwendet.

4.2.4.2.3 Klassifizieren mit Einzelwort-Features mit dem Naive-Bayes-Algorithmus

Auch in diesem Fall wird das gleiche Verfahren zum Klassifizieren auf die Lewis-Feature-Vektoren angewendet wie zuvor auf die Feature-Vektoren, die auf den TSF-Features basieren. Der Unterschied besteht ebenfalls in den Pfaden zum Speicher- oder Ladeverzeichnis.

4.2.4.2.4 Klassifizieren mit Einzelwort-Features mit dem Support-Vector-Machine-Algorithmus

Auch in diesem Fall wird das gleiche Verfahren zum Klassifizieren auf die Lewis-Feature-Vektoren angewendet wie zuvor auf die Feature-Vektoren, die auf den TSF-Features basieren. Der Unterschied besteht darin, dass SVM^{multiclass} beim Training und beim Klassifizieren die entsprechenden wortbasierten Eingabedateien erhält. Die Parametereinstellung sowie der Kernel sind gleich.

4.2.5 Evaluation des Klassifizierens mit Einzelwort-Features

Bei der Evaluation der Ergebnisse des Klassifizierens mit Hilfe der Lewis-Feature-Vektoren kommen die gleichen Verfahren zur Anwendung wie bei der Evaluation der Klassifizierungen, die auf den aus den Suffix Arrays abgeleiteten Feature-Vektoren basieren. Die Beschreibung der Vorgehensweise ist in Kapitel 4.1.10 zu finden. Der einzige Unterschied besteht darin, die entsprechend serialisierten Objekte aus den anderen Verzeichnispfaden einzulesen.
Bei der SVM wird die Operation `prepareEvaluationSubsetsTestSVMMulticlass-Lewis` aufgerufen, die derjenigen im Fall der Vektoren, die auf TSF-Features basieren, entspricht. Hier werden lediglich die entsprechenden Ergebnisse der SVM bezogen auf die Lewis-Vektoren verarbeitet. Ansonsten entspricht der Evaluationsablauf dem in Kapitel 4.1.10.1.2 beschriebenen.
Auch im Fall der Evaluation mit dem F_1-Maß erfolgt die gleiche Verarbeitung wie zuvor beschrieben.

4.3 Durchgeführte Klassifizierungsexperimente

4.3.1 Definition eines Klassifizierungsexperiments

Im Rahmen dieser Arbeit wird das entwickelte Verfahren der Ermittlung von TSF-Features für Dokumente über Suffix Arrays mit Hilfe von Experimenten überprüft. Dabei wird das gleiche Experiment, zum einen für die mit dem entwickelten Verfahren erzeugten Features, und zum anderen mit den von D.D. Lewis zur Verfügung gestellten einzelwortbasierten Features durchgeführt und die Ergebnisse werden miteinander verglichen. Für jedes Experiment erfolgt eine Beschreibung nach dem folgenden Schema:

- ID
- Bezeichnung

- Trainingsdaten
- Testdaten
- Ähnlichkeit
- Algorithmen
- Baseline
- Evaluation
- Ergebnisse
- Interpretation der Ergebnisse

Diese „Überschriften“ werden für jedes durchgeführte Experiment ausgefüllt. Ihr Inhalt genügt den folgenden Definitionen:

- ID

Die ID eines Experiments ist eine Kennung, die es innerhalb der vorliegenden Arbeit eindeutig identifiziert. Sie setzt sich zusammen aus:

* einer Bezeichnung, die angibt, ob es sich um ein Klassifizierungs- (Klass) oder Clusterexperiment (Clus) handelt,
* einer aufsteigenden Nummer beginnend bei 1,
* einer Bezeichnung, die angibt, um welche Klassenfamilie es in dem Experiment geht: I entspricht Industry, R entspricht Region und T entspricht Topic sowie
* einer Bezeichnung, die angibt, dass die Ähnlichkeit zwischen den Dokumenten auf Vektoren beruht, also ein V.

Als Beispiel: Experiment 2 mit vektorbasierter Industry-Klassifizierung würde durch Klass2IV bezeichnet.

- Bezeichnung

Die Bezeichnung eines Experiments beschreibt, um welches Experiment es sich handelt. Sie ist als Ergänzung zur ID zu verstehen. Hier würde das Beispiel aus dem Absatz über die ID wie folgt lauten: Vektorbasierte Single-label-Industry-Klassifizierung von Testdaten.

- Trainingsdaten

Hier erfolgt eine Beschreibung der in dem Experiment verwendeten Trainingsdaten. Darunter fallen die Anzahl der Trainingsteilmengen, ihre Bezeichnungen, die Anzahl der jeweils enthaltenen Trainingsdokumente und die Klassenzuordnung dieser Trainingsdokumente im Hinblick auf die Frage, ob genau eine oder mehrere Klassen der Klassenfamilie zugeordnet sind. So würde man beispielsweise angeben, dass man mit einer Trainingsteilmenge mit der Bezeichnung Tr1 mit 20 enthaltenen Dokumenten, die genau einer Region-Klasse zugeordnet sind, die Klassifizierer trainiert.

- Testdaten

Diese Beschreibung ist die gleiche, die für die Trainingsdaten erstellt wird, jedoch bezogen auf die verwendeten Testdaten.

- Ähnlichkeit

Unter diesem Punkt wird angegeben, wie die Ähnlichkeit zwischen den Dokumenten für das Experiment definiert ist. Eine Möglichkeit ist die vektorbasierte Ähnlichkeit. Das bedeutet bspw., alle verwendeten Algorithmen zum Klassifizieren verwenden Vektoren der Dokumente als Darstellung und für alle Operationen.

- Algorithmen

Hier werden die Algorithmen angegeben, mit denen das Experiment durchgeführt und die Ergebnisse erzeugt werden. Ein Experiment kann in der gleichen „Konfiguration“ der Daten, der betrachteten Klassenfamilie und der Ähnlichkeit mit mehreren Algorithmen durchgeführt werden. Beispielsweise könnte ein Klassifizierungs-Experiment das Klassifizieren der Testdaten basierend auf den Trainingsdaten einerseits mit einem Naive-Bayes- und andererseits mit einem Decision-Tree-Algorithmus vornehmen.

- Baseline

Die erstellten Klassifikationen werden evaluiert und können dann miteinander verglichen werden. Es findet also bei der Auswertung der Qualität nur ein Vergleich zwischen den beiden Verfahren statt und keine Beurteilung der absoluten Qualität. Um dennoch zu zeigen, dass die Qualität der erzeugten Klassifikationen besser ist als eine zufällige Zuordnung der Dokumente zu den vorgegebenen Klassen, wird zusätzlich ein Vergleich mit einer so genannten *Baseline* durchgeführt. Unter Baseline

versteht man laut Bortz u.a. (2002)[1] den Normalfall oder die Häufigkeit für das Auftreten eines Ereignisses, die „objektiv“ ist. Übertragen auf die Klassifizierung von Dokumenten ist die Baseline in diesem Fall die zufällige Zuordnung der Dokumente zu den vorgegebenen Klassen, also eine Zuordnung, die getroffen wird, wenn man keine zusätzlichen Informationen hat, wie die Dokumente zugeordnet werden sollten. Als Baseline kommen in der vorliegenden Arbeit zwei Methoden zum Einsatz:

1. Randomisierte Zuordnung aller Testdokumente zu allen möglichen Klassen der Trainingsdaten
   In diesem Fall wird angenommen, dass die Dokumente jeder der Klassen, die in der betrachteten Trainingsteilmenge vorkommen, zugeordnet werden können, und genau dies wird auch durchgeführt. Man legt also eine Wahrscheinlichkeit von $\frac{1}{\text{Anzahl der Klassen der Trainingsdokumente}}$ zu Grunde. Diese Wahrscheinlichkeit bildet die erste Baseline.

2. Zuordnung aller Testdokumente zu der Klasse, der die meisten Trainingsdokumente zugeordnet sind
   Diesen Fall nennt man die *„majority-class baseline“*[2]. Dafür wird aus den Trainingsdokumenten die Klasse ermittelt, der die meisten Trainingsdokumente zugeordnet sind. Dieser Klasse werden alle Testdokumente zugeordnet[3] und diese Klassifikation evaluiert. Das Ergebnis der Evaluation bildet die zweite Baseline.[4]

- Evaluation

Dieser Punkt beschreibt, wie die Evaluation der Ergebnisse vorgenommen wird. Dahinter verbirgt sich die Angabe, welches Evaluationsmaß für das Experiment sowohl in Bezug auf TSF-Feature-Vektoren als auch in Bezug auf Lewis-Feature-Vektoren verwendet wird. Eine Möglichkeit wäre beispielsweise anzugeben, dass das F-Maß[5] als Evaluationsmaß verwendet wird.

1 Vgl. Bortz u.a. (2002), S. 184, 581.
2 Wilson u.a. (2011), S. 421, Kursivschreibung durch die Verfasserin.
3 Vgl. Irani u.a. (2010), S. 5 (Seitennummerierung durch die Verfasserin, da Internetdokument).
4 Eine genaue Beschreibung der Baselines für die einzelnen Experimente können den Experimentbeschreibungen entnommen werden.
5 Siehe Kapitel 2.5.2.2 dieser Arbeit.

- Ergebnisse

An dieser Stelle der Experimentbeschreibung werden die erreichten Ergebnisse vorgestellt. Dabei handelt es sich um eine Gegenüberstellung der auf den TSF-Feature-Vektoren basierenden Ergebnisse und der Ergebnisse, die auf den Lewis-Feature-Vektoren basieren. Die Ergebnisse beinhalten den berechneten Wert des verwendeten Evaluationsmaßes. Beispielsweise könnte das oben genannte F-Maß einen Wert von 0,4 für die Suffix-Arrays-basierte Klassifizierung liefern und einen Wert von 0,35 für die Lewis-Feature-Vektor-basierte Klassifizierung der Testmenge Te4, wobei der Naive-Bayes-Klassifizierer mit der Trainingsmenge Tr2 trainiert wurde.

- Interpretation der Ergebnisse

Den Abschluss einer jeden Experimentbeschreibung bildet eine Interpretation der erreichten Ergebnisse. Darunter fällt beispielsweise der Vergleich zwischen den Ergebnissen, basierend auf Suffix Arrays, und denen, basierend auf den Lewis-Vektoren, in Bezug auf das durchgeführte Experiment.

4.3.2 Klassifizierungsexperimente

4.3.2.1 Klassifizierungsexperimente mit dem korrekten Anteil als Evaluationsmaß

4.3.2.1.1 Experiment 1

- ID
  Klass1RV
- Bezeichnung
  vektorbasiertes Single-label-Region-Klassifizieren von Testdaten der Reuters-Daten RCV1-v2
- Trainingsdaten
  - 3 randomisiert zusammengestellte Teilmengen der Trainingsdaten des RCV1-v2 mit jeweils 2.000 Dokumenten: Tr1, Tr2, Tr3
  - Dokumente von Tr1 bis Tr3 gehören zu genau einer Region-Klasse
- Testdaten
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 2.000 Dokumenten: Te1, Te2
  - Dokumente in Te1 und Te2 haben beliebige Klassenzugehörigkeiten[1]

1 Zur Erinnerung: Jedes Dokument muss mindestens einer Region-Klasse zugeordnet sein.

  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 10.000 Dokumenten: Te3, Te4
  - Dokumente in Te3 und Te4 haben beliebige Klassenzugehörigkeiten
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 2.000 Dokumenten: Te5, Te6
  - Dokumente in Te5 und Te6 gehören zu genau einer Region-Klasse
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 10.000 Dokumenten: Te7, Te8
  - Dokumente in Te7 und Te8 gehören zu genau einer Region-Klasse

- Ähnlichkeit
  vektorbasiert

- Algorithmen
  - Naive Bayes
  - Decision Tree
  - k-Nearest-Neighbour mit $k = 10$, 20 und 200 Nachbarn mit Cosinus zur Ähnlichkeitsberechnung zwischen den Vektoren
  - Support Vector Machine SVM^{multiclass} mit Parameter $C = 1,0$ und linearem Kernel

- Baseline
  - randomisierte Zuordnung
  - Majority-Class-Zuordnung zu den zwei Klassen mit den meisten Trainingsdokumenten

- Evaluation
  Anteil der richtig klassifizierten Testdaten pro Testteilmenge an der Anzahl der in der Testteilmenge vorhandenen Dokumente

- Ergebnisse[1]

  Für das Klassifizieren der Testmengen Te1 bis Te8 mit dem Naive-Bayes-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.1, 4.2 und 4.3, zu sehen auf S. 311 bis 312.

  Für das Klassifizieren der Testmengen Te1 bis Te8 mit dem Decision-Tree-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.4, 4.5 und 4.6, zu sehen auf S. 313 bis 314.

  Für das Klassifizieren der Testmengen Te1 bis Te8 mit dem k-Nearest-Neighbour-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3 und 10, 20 und 200 Nachbarn, ergeben sich die Ergebnisse aus den Tabellen 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 4.13, 4.14 und 4.15, zu sehen auf S. 315 bis 320.

  Für das Klassifizieren der Testmengen Te1 bis Te8 mit dem Support-Vector-Machine-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.16, 4.17 und 4.18, zu sehen auf S. 321 bis 322.

Tabelle 4.1: Ergebnisse des Experiments Klass1RV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	818	1182	40,90
	Region	2	2000	nein	750	1250	37,50
	Region	3	10000	nein	4030	5970	40,30
	Region	4	10000	nein	3942	6058	39,42
	Region	5	2000	ja	811	1189	40,55
	Region	6	2000	ja	826	1174	41,30
	Region	7	10000	ja	4124	5876	41,24
	Region	8	10000	ja	4127	5873	41,27
Lewis-Feature-Vektoren	Region	1	2000	nein	245	1755	12,25
	Region	2	2000	nein	266	1734	13,30
	Region	3	10000	nein	1348	8652	13,48
	Region	4	10000	nein	1283	8717	12,83
	Region	5	2000	ja	226	1774	11,30
	Region	6	2000	ja	200	1800	10,00
	Region	7	10000	ja	1071	8929	10,71
wird auf der nächsten Seite fortgesetzt							

1 In allen Tabellen werden die Ergebnisse auf zwei Stellen nach dem Komma gerundet. Alle Klassifizierer sind im abschließenden Vergleich durch ihre Abkürzungen aufgelistet: NB für Naive Bayes, DT für Decision Tree, kNN für k-Nearest-Neighbour und SVM für Support Vector Machine.

Tabelle 4.1: Ergebnisse des Experiments Klass1RV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr1 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Region	8	10000	ja	1093	8907	10,93

Tabelle 4.2: Ergebnisse des Experiments Klass1RV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	805	1195	40,25
	Region	2	2000	nein	739	1261	36,95
	Region	3	10000	nein	3964	6036	39,64
	Region	4	10000	nein	3924	6076	39,24
	Region	5	2000	ja	804	1196	40,20
	Region	6	2000	ja	823	1177	41,15
	Region	7	10000	ja	4099	5901	40,99
	Region	8	10000	ja	4111	5889	41,11
Lewis-Feature-Vektoren	Region	1	2000	nein	255	1745	12,75
	Region	2	2000	nein	267	1733	13,35
	Region	3	10000	nein	1378	8622	13,78
	Region	4	10000	nein	1332	8668	13,32
	Region	5	2000	ja	245	1755	12,25
	Region	6	2000	ja	208	1792	10,40
	Region	7	10000	ja	1146	8854	11,46
	Region	8	10000	ja	1128	8872	11,28

Tabelle 4.3: Ergebnisse des Experiments Klass1RV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	800	1200	40,00
	Region	2	2000	nein	742	1258	37,10
	Region	3	10000	nein	3897	6103	38,97
	Region	4	10000	nein	3835	6165	38,35
wird auf der nächsten Seite fortgesetzt							

Tabelle 4.3: Ergebnisse des Experiments Klass1RV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr3 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Region	5	2000	ja	787	1213	39,35
	Region	6	2000	ja	808	1192	40,40
	Region	7	10000	ja	4054	5946	40,54
	Region	8	10000	ja	4056	5944	40,56
Lewis-Feature-Vektoren	Region	1	2000	nein	565	1435	28,25
	Region	2	2000	nein	496	1504	24,80
	Region	3	10000	nein	2663	7337	26,63
	Region	4	10000	nein	2710	7290	27,10
	Region	5	2000	ja	508	1492	25,40
	Region	6	2000	ja	522	1478	26,10
	Region	7	10000	ja	2603	7397	26,03
	Region	8	10000	ja	2628	7372	26,28

Tabelle 4.4: Ergebnisse des Experiments Klass1RV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	1172	828	58,60
	Region	2	2000	nein	1113	887	55,65
	Region	3	10000	nein	5746	4254	57,46
	Region	4	10000	nein	5709	4291	57,09
	Region	5	2000	ja	1184	816	59,20
	Region	6	2000	ja	1236	764	61,80
	Region	7	10000	ja	6028	3972	60,28
	Region	8	10000	ja	6045	3955	60,45
Lewis-Feature-Vektoren	Region	1	2000	nein	1124	876	56,20
	Region	2	2000	nein	1047	953	52,35
	Region	3	10000	nein	5458	4542	54,58
	Region	4	10000	nein	5313	4687	53,13
	Region	5	2000	ja	1140	860	57,00
	Region	6	2000	ja	1119	881	55,95
	Region	7	10000	ja	5619	4381	56,19

wird auf der nächsten Seite fortgesetzt

Tabelle 4.4: Ergebnisse des Experiments Klass1RV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr1 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Region	8	10000	ja	5624	4376	56,24

Tabelle 4.5: Ergebnisse des Experiments Klass1RV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	1192	808	59,60
	Region	2	2000	nein	1155	845	57,75
	Region	3	10000	nein	5949	4051	59,49
	Region	4	10000	nein	5832	4168	58,32
	Region	5	2000	ja	1218	782	60,90
	Region	6	2000	ja	1270	730	63,50
	Region	7	10000	ja	6124	3876	61,24
	Region	8	10000	ja	6151	3849	61,51
Lewis-Feature-Vektoren	Region	1	2000	nein	1015	985	50,75
	Region	2	2000	nein	957	1043	47,85
	Region	3	10000	nein	4925	5075	49,25
	Region	4	10000	nein	4811	5189	48,11
	Region	5	2000	ja	1019	981	50,95
	Region	6	2000	ja	1031	969	51,55
	Region	7	10000	ja	5139	4861	51,39
	Region	8	10000	ja	5211	4789	52,11

Tabelle 4.6: Ergebnisse des Experiments Klass1RV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	1155	845	57,75
	Region	2	2000	nein	1074	926	53,70
	Region	3	10000	nein	5753	4247	57,53
	Region	4	10000	nein	5638	4362	56,38
wird auf der nächsten Seite fortgesetzt							

Tabelle 4.6: Ergebnisse des Experiments Klass1RV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr3 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Region	5	2000	ja	1178	822	58,90
	Region	6	2000	ja	1221	779	61,05
	Region	7	10000	ja	5970	4030	59,70
	Region	8	10000	ja	6014	3986	60,14
Lewis-Feature-Vektoren	Region	1	2000	nein	955	1045	47,75
	Region	2	2000	nein	906	1094	45,30
	Region	3	10000	nein	4727	5273	47,27
	Region	4	10000	nein	4599	5401	45,99
	Region	5	2000	ja	983	1017	49,15
	Region	6	2000	ja	971	1029	48,55
	Region	7	10000	ja	4878	5122	48,78
	Region	8	10000	ja	4844	5156	48,44

Tabelle 4.7: Ergebnisse des Experiments Klass1RV für den 10NN-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	1319	681	65,95
	Region	2	2000	nein	1312	688	65,60
	Region	3	10000	nein	6692	3308	66,92
	Region	4	10000	nein	6658	3342	66,58
	Region	5	2000	ja	1420	580	71,00
	Region	6	2000	ja	1418	582	70,90
	Region	7	10000	ja	7049	2951	70,49
	Region	8	10000	ja	7047	2953	70,47
Lewis-Feature-Vektoren	Region	1	2000	nein	439	1561	21,95
	Region	2	2000	nein	437	1563	21,85
	Region	3	10000	nein	2181	7819	21,81
	Region	4	10000	nein	2050	7950	20,50
	Region	5	2000	ja	488	1512	24,40
	Region	6	2000	ja	520	1480	26,00
	Region	7	10000	ja	2430	7570	24,30

wird auf der nächsten Seite fortgesetzt

Tabelle 4.7: Ergebnisse des Experiments Klass1RV für den 10NN-Klassifizierer für Trainingsmenge Tr1 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Region	8	10000	ja	2527	7473	25,27

Tabelle 4.8: Ergebnisse des Experiments Klass1RV für den 20NN-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	1331	669	66,55
	Region	2	2000	nein	1317	683	65,85
	Region	3	10000	nein	6761	3239	67,61
	Region	4	10000	nein	6699	3301	66,99
	Region	5	2000	ja	1416	584	70,80
	Region	6	2000	ja	1407	593	70,35
	Region	7	10000	ja	7051	2949	70,51
	Region	8	10000	ja	7040	2960	70,40
Lewis-Feature-Vektoren	Region	1	2000	nein	875	1125	43,75
	Region	2	2000	nein	868	1132	43,40
	Region	3	10000	nein	4397	5603	43,97
	Region	4	10000	nein	4310	5690	43,10
	Region	5	2000	ja	917	1083	45,85
	Region	6	2000	ja	924	1076	46,20
	Region	7	10000	ja	4556	5444	45,56
	Region	8	10000	ja	4629	5371	46,29

Tabelle 4.9: Ergebnisse des Experiments Klass1RV für den 200NN-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	961	1039	48,05
	Region	2	2000	nein	902	1098	45,10
	Region	3	10000	nein	4748	5252	47,48
	Region	4	10000	nein	4611	5389	46,11
wird auf der nächsten Seite fortgesetzt							

Tabelle 4.9: Ergebnisse des Experiments Klass1RV für den 200NN-Klassifizierer für Trainingsmenge Tr1 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Region	5	2000	ja	957	1043	47,85
	Region	6	2000	ja	970	1030	48,50
	Region	7	10000	ja	4830	5170	48,30
	Region	8	10000	ja	4796	5204	47,96
Lewis-Feature-Vektoren	Region	1	2000	nein	761	1239	38,05
	Region	2	2000	nein	707	1293	35,35
	Region	3	10000	nein	3698	6302	36,98
	Region	4	10000	nein	3652	6348	36,52
	Region	5	2000	ja	755	1245	37,75
	Region	6	2000	ja	756	1244	37,80
	Region	7	10000	ja	3777	6223	37,77
	Region	8	10000	ja	3832	6168	38,32

Tabelle 4.10: Ergebnisse des Experiments Klass1RV für den 10NN-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	1331	669	66,55
	Region	2	2000	nein	1357	643	67,85
	Region	3	10000	nein	6838	3162	68,38
	Region	4	10000	nein	6835	3165	68,35
	Region	5	2000	ja	1455	545	72,75
	Region	6	2000	ja	1449	551	72,45
	Region	7	10000	ja	7150	2850	71,50
	Region	8	10000	ja	7114	2886	71,14
Lewis-Feature-Vektoren	Region	1	2000	nein	569	1431	28,45
	Region	2	2000	nein	578	1422	28,90
	Region	3	10000	nein	3005	6995	30,05
	Region	4	10000	nein	2935	7065	29,35
	Region	5	2000	ja	608	1392	30,40
	Region	6	2000	ja	616	1384	30,80
	Region	7	10000	ja	3001	6999	30,01

wird auf der nächsten Seite fortgesetzt

Tabelle 4.10: Ergebnisse des Experiments Klass1RV für den 10NN-Klassifizierer für Trainingsmenge Tr2 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Region	8	10000	ja	3035	6965	30,35

Tabelle 4.11: Ergebnisse des Experiments Klass1RV für den 20NN-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	1344	656	67,20
	Region	2	2000	nein	1338	662	66,90
	Region	3	10000	nein	6862	3138	68,62
	Region	4	10000	nein	6805	3195	68,05
	Region	5	2000	ja	1444	556	72,20
	Region	6	2000	ja	1444	556	72,20
	Region	7	10000	ja	7141	2859	71,41
	Region	8	10000	ja	7085	2915	70,85
Lewis-Feature-Vektoren	Region	1	2000	nein	559	1441	27,95
	Region	2	2000	nein	557	1443	27,85
	Region	3	10000	nein	2971	7029	29,71
	Region	4	10000	nein	2886	7114	28,86
	Region	5	2000	ja	600	1400	30,00
	Region	6	2000	ja	609	1391	30,45
	Region	7	10000	ja	2923	7077	29,23
	Region	8	10000	ja	2977	7023	29,77

Tabelle 4.12: Ergebnisse des Experiments Klass1RV für den 200NN-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	944	1056	47,20
	Region	2	2000	nein	884	1116	44,20
	Region	3	10000	nein	4660	5340	46,60
	Region	4	10000	nein	4547	5453	45,47
wird auf der nächsten Seite fortgesetzt							

Tabelle 4.12: Ergebnisse des Experiments Klass1RV für den 200NN-Klassifizierer für Trainingsmenge Tr2 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Region	5	2000	ja	942	1058	47,10
	Region	6	2000	ja	949	1051	47,45
	Region	7	10000	ja	4719	5281	47,19
	Region	8	10000	ja	4732	5268	47,32
Lewis-Feature-Vektoren	Region	1	2000	nein	744	1256	37,20
	Region	2	2000	nein	693	1307	34,65
	Region	3	10000	nein	3624	6376	36,24
	Region	4	10000	nein	3565	6435	35,65
	Region	5	2000	ja	720	1280	36,00
	Region	6	2000	ja	745	1255	37,25
	Region	7	10000	ja	3713	6287	37,13
	Region	8	10000	ja	3726	6274	37,26

Tabelle 4.13: Ergebnisse des Experiments Klass1RV für den 10NN-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	1327	673	66,35
	Region	2	2000	nein	1314	686	65,70
	Region	3	10000	nein	6704	3296	67,04
	Region	4	10000	nein	6692	3308	66,92
	Region	5	2000	ja	1418	582	70,90
	Region	6	2000	ja	1429	571	71,45
	Region	7	10000	ja	7054	2946	70,54
	Region	8	10000	ja	7021	2979	70,21
Lewis-Feature-Vektoren	Region	1	2000	nein	863	1137	43,15
	Region	2	2000	nein	857	1143	42,85
	Region	3	10000	nein	4319	5681	43,19
	Region	4	10000	nein	4284	5716	42,84
	Region	5	2000	ja	924	1076	46,20
	Region	6	2000	ja	917	1083	45,85
	Region	7	10000	ja	4589	5411	45,89

wird auf der nächsten Seite fortgesetzt

Tabelle 4.13: Ergebnisse des Experiments Klass1RV für den 10NN-Klassifizierer für Trainingsmenge Tr3 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Region	8	10000	ja	4575	5425	45,75

Tabelle 4.14: Ergebnisse des Experiments Klass1RV für den 20NN-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	1322	678	66,10
	Region	2	2000	nein	1298	702	64,90
	Region	3	10000	nein	6737	3263	67,37
	Region	4	10000	nein	6658	3342	66,58
	Region	5	2000	ja	1411	589	70,55
	Region	6	2000	ja	1418	582	70,90
	Region	7	10000	ja	7000	3000	70,00
	Region	8	10000	ja	6984	3016	69,84
Lewis-Feature-Vektoren	Region	1	2000	nein	872	1128	43,60
	Region	2	2000	nein	860	1140	43,00
	Region	3	10000	nein	4307	5693	43,07
	Region	4	10000	nein	4274	5726	42,74
	Region	5	2000	ja	909	1091	45,45
	Region	6	2000	ja	934	1066	46,70
	Region	7	10000	ja	4557	5443	45,57
	Region	8	10000	ja	4546	5454	45,46

Tabelle 4.15: Ergebnisse des Experiments Klass1RV für den 200NN-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	951	1049	47,55
	Region	2	2000	nein	907	1093	45,35
	Region	3	10000	nein	4679	5321	46,79
	Region	4	10000	nein	4570	5430	45,70
wird auf der nächsten Seite fortgesetzt							

Tabelle 4.15: Ergebnisse des Experiments Klass1RV für den 200NN-Klassifizierer für Trainingsmenge Tr3 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Region	5	2000	ja	951	1049	47,55
	Region	6	2000	ja	946	1054	47,30
	Region	7	10000	ja	4736	5264	47,36
	Region	8	10000	ja	4743	5257	47,43
Lewis-Feature-Vektoren	Region	1	2000	nein	733	1267	36,65
	Region	2	2000	nein	695	1305	34,75
	Region	3	10000	nein	3599	6401	35,99
	Region	4	10000	nein	3542	6458	35,42
	Region	5	2000	ja	726	1274	36,30
	Region	6	2000	ja	739	1261	36,95
	Region	7	10000	ja	3699	6301	36,99
	Region	8	10000	ja	3715	6285	37,15

Tabelle 4.16: Ergebnisse des Experiments Klass1RV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	1274	726	63,70
	Region	2	2000	nein	1244	756	62,20
	Region	3	10000	nein	6399	3601	63,99
	Region	4	10000	nein	6288	3712	62,88
	Region	5	2000	ja	1297	703	64,85
	Region	6	2000	ja	1313	687	65,65
	Region	7	10000	ja	6515	3485	65,15
	Region	8	10000	ja	6594	3406	65,94
Lewis-Feature-Vektoren	Region	1	2000	nein	1122	878	56,10
	Region	2	2000	nein	1070	930	53,50
	Region	3	10000	nein	5558	4442	55,58
	Region	4	10000	nein	5440	4560	54,40
	Region	5	2000	ja	1117	883	55,85
	Region	6	2000	ja	1104	896	55,20
	Region	7	10000	ja	5673	4327	56,73

wird auf der nächsten Seite fortgesetzt

Tabelle 4.16: Ergebnisse des Experiments Klass1RV für Support Vector Machine für Trainingsmenge Tr1 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Region	8	10000	ja	5614	4386	56,14

Tabelle 4.17: Ergebnisse des Experiments Klass1RV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	1247	753	62,35
	Region	2	2000	nein	1235	765	61,75
	Region	3	10000	nein	6287	3713	62,87
	Region	4	10000	nein	6214	3786	62,14
	Region	5	2000	ja	1278	722	63,90
	Region	6	2000	ja	1316	684	65,80
	Region	7	10000	ja	6492	3508	64,92
	Region	8	10000	ja	6465	3535	64,65
Lewis-Feature-Vektoren	Region	1	2000	nein	1132	868	56,60
	Region	2	2000	nein	1123	877	56,15
	Region	3	10000	nein	5755	4245	57,55
	Region	4	10000	nein	5591	4409	55,91
	Region	5	2000	ja	1153	847	57,65
	Region	6	2000	ja	1204	796	60,20
	Region	7	10000	ja	5897	4103	58,97
	Region	8	10000	ja	5857	4143	58,57

Tabelle 4.18: Ergebnisse des Experiments Klass1RV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Region	1	2000	nein	1225	775	61,25
	Region	2	2000	nein	1195	805	59,75
	Region	3	10000	nein	6139	3861	61,39
	Region	4	10000	nein	6014	3986	60,14
wird auf der nächsten Seite fortgesetzt							

Tabelle 4.18: Ergebnisse des Experiments Klass1RV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr3 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Region	5	2000	ja	1250	750	62,50
	Region	6	2000	ja	1299	701	64,95
	Region	7	10000	ja	6337	3663	63,37
	Region	8	10000	ja	6372	3628	63,72
Lewis-Feature-Vektoren	Region	1	2000	nein	1074	926	53,70
	Region	2	2000	nein	1038	962	51,90
	Region	3	10000	nein	5264	4736	52,64
	Region	4	10000	nein	5185	4815	51,85
	Region	5	2000	ja	1070	930	53,50
	Region	6	2000	ja	1093	907	54,65
	Region	7	10000	ja	5480	4520	54,80
	Region	8	10000	ja	5451	4549	54,51

- Interpretation der Ergebnisse
  - Einführung
    Im Folgenden werden die vorgestellten Ergebnisse für die auf den Suffix Arrays basierenden Feature-Vektoren und für die Lewis-Feature-Vektoren einzeln pro Algorithmus miteinander verglichen. Abschließend wird ein allgemeiner Vergleich zwischen den zwei Verfahren bezogen auf das gesamte Experiment über alle Algorithmen und Trainingsteilmengen durchgeführt.
  - Ergebnisvergleich Naive Bayes
    Bildet man die Ergebnisse, die mit dem Naive-Bayes-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man die Abbildung 4.66 auf S. 324.[1] Die abgebildeten Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features.

1 Die Abkürzungen in den Diagrammen bedeuten für dieses Experiment Folgendes: Baseline wird mit „BL“ abgekürzt, „USA“ steht für den Klassennamen der Klasse mit den meisten Trainingsdokumenten und „%“ steht für die Baseline mit der randomisierten Zuordnung.

Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 23,06 Prozentpunkten zwischen den erreichten Ergebnissen mit den TSF-Features und den Einzelwort-Features. Der minimale Abstand betrug 11,25 Prozentpunkte und der maximale Abstand 31,3 Prozentpunkte.

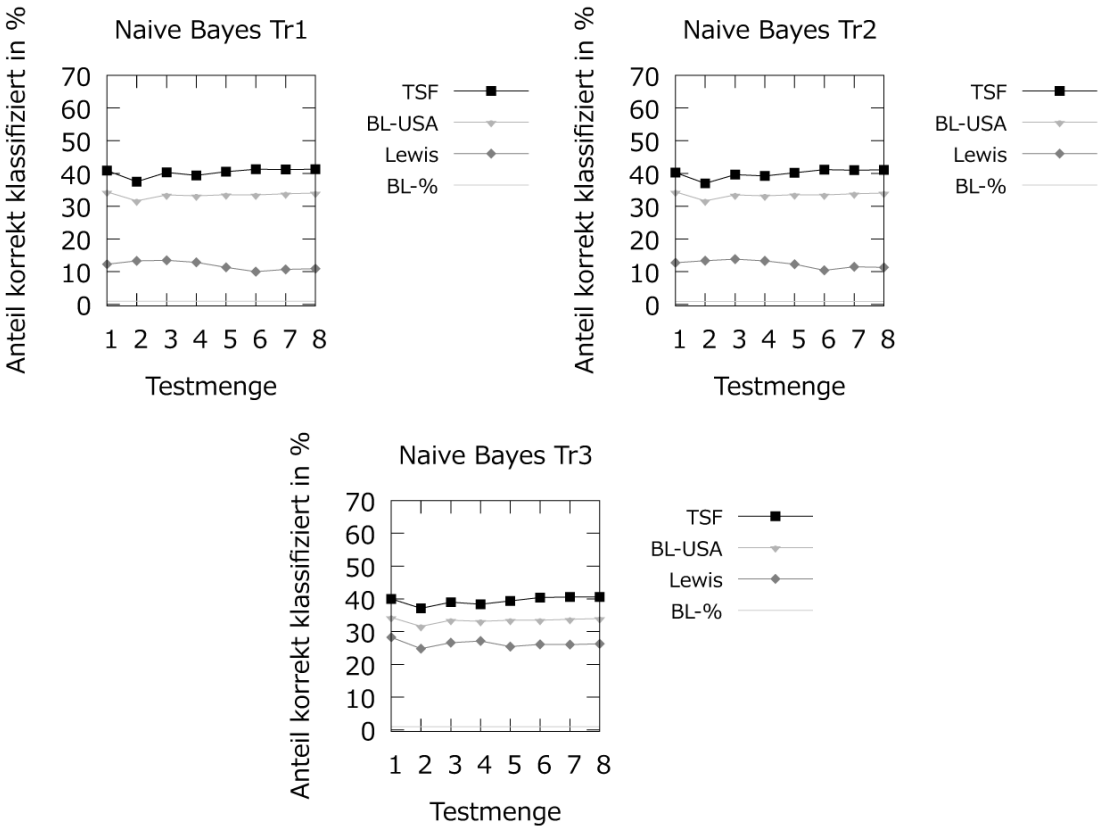


Abbildung 4.66: Ergebnis der Evaluation der Klassifizierung durch den Naive-Bayes-Klassifizierer für das Experiment Klass1RV

Die Trainingsteilmengen bestehen alle drei aus Dokumenten, die nur eine Klassenzuordnung der betrachteten Klassenfamilie Region haben. Um Unterschiede feststellen zu können, wurden in den Testteilmengen zwei verschiedene Auswahlkriterien bezüglich der Klassenzuordnungen der Dokumente getroffen. So ist der erste Teil dieser Teilmengen, die Te1 bis Te4, keiner Einschränkung unterworfen hinsichtlich der Anzahl der Klassenzuordnungen der betrachteten Klassenfamilie, die die enthaltenen Dokumente haben. Das bedeutet, die Dokumente haben mindestens eine Region-Zuordnung, können

aber auch mehrere haben. Der zweite Teil dieser Teilmengen, Te5 bis Te8 dagegen enthält nur Testdokumente, die *genau eine* Region-Zuordnung haben. Die Annahme ist, dass, wenn mehr als eine Klasse korrekt sein kann, eine höhere Wahrscheinlichkeit besteht, dass eine der korrekten Klassen zugeordnet wurde.[1] Also sollte die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen als für Te5 bis Te8. Bei den TSF-Feature-Vektoren trifft diese Annahme für die Klassenfamilie Region für den Naive-Bayes-Algorithmus nicht zu. Bei den einzelwortbasierten Vektoren trifft sie jedoch schon zu.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal um die 40% richtig klassifizierter Dokumente erreicht wird. Für die TSF-Feature-Vektoren gilt, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit einer zufälligen Zuordnung der Testdokumente zu den zuordenbaren Klassen, d.h., es werden bessere Ergebnisse erreicht als mit der Baseline-Klassifizierung. Das gilt für die einzelwortbasierten Feature-Vektoren jedoch annähernd nur für die Trainingsteilmenge Tr3. Bei den anderen beiden Trainingsteilmengen wäre eine höhere Qualität erreicht worden, wenn die Testdokumente der größten Klasse der Trainingsteilmenge direkt zugeordnet worden wären, ohne einen Klassifizierer über die zuzuordnende Klasse entscheiden zu lassen. Das bedeutet, dass auch die Zusammensetzung der Trainings- und Testteilmengen die Qualität des Ergebnisses beeinflusst.

– Ergebnisvergleich Decision Tree
  Bildet man die Ergebnisse, die mit dem Decision-Tree-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Abbildung 4.67 auf S. 326. Die abgebildeten Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Test-

1 Stellt man sich als Grundannahme die Zuordnung von mindestens einer Klasse als „Ziehen“ mindestens einer Klasse aus dem „Topf“ der möglichen Klassen vor, so ergibt sich für *eine* von Reuters zugeordnete Klasse eine Wahrscheinlichkeit von $\frac{1}{|L|}$. Für g (mit $g > 1$) von Reuters zugeordnete Klassen ergibt sich dagegen eine Wahrscheinlichkeit von $\frac{g}{|L|}$, also eine höhere Wahrscheinlichkeit. Da in der hier durchgeführten Evaluation pro Dokument eine korrekte Klassifizierung vorliegt, sobald die zugeordnete Klasse einer der durch Reuters zugeordneten Klassen entspricht, ist die Grundannahme, dass die Wahrscheinlichkeit einer korrekten Klassifizierung bei mehreren durch Reuters zugeordneten Klassen höher liegt.

teilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 8,05 Prozentpunkten zwischen den Ergebnissen mit den TSF-Features und den Ergebnissen der Einzelwort-Features. Der minimale Abstand betrug 2,2 Prozentpunkte und der maximale Abstand 12,5 Prozentpunkte. Im Gegensatz zum Naive-Bayes-Algorithmus liegen die Ergebnisse des Decision-Tree-Algorithmus für beide Verfahren nah beieinander.

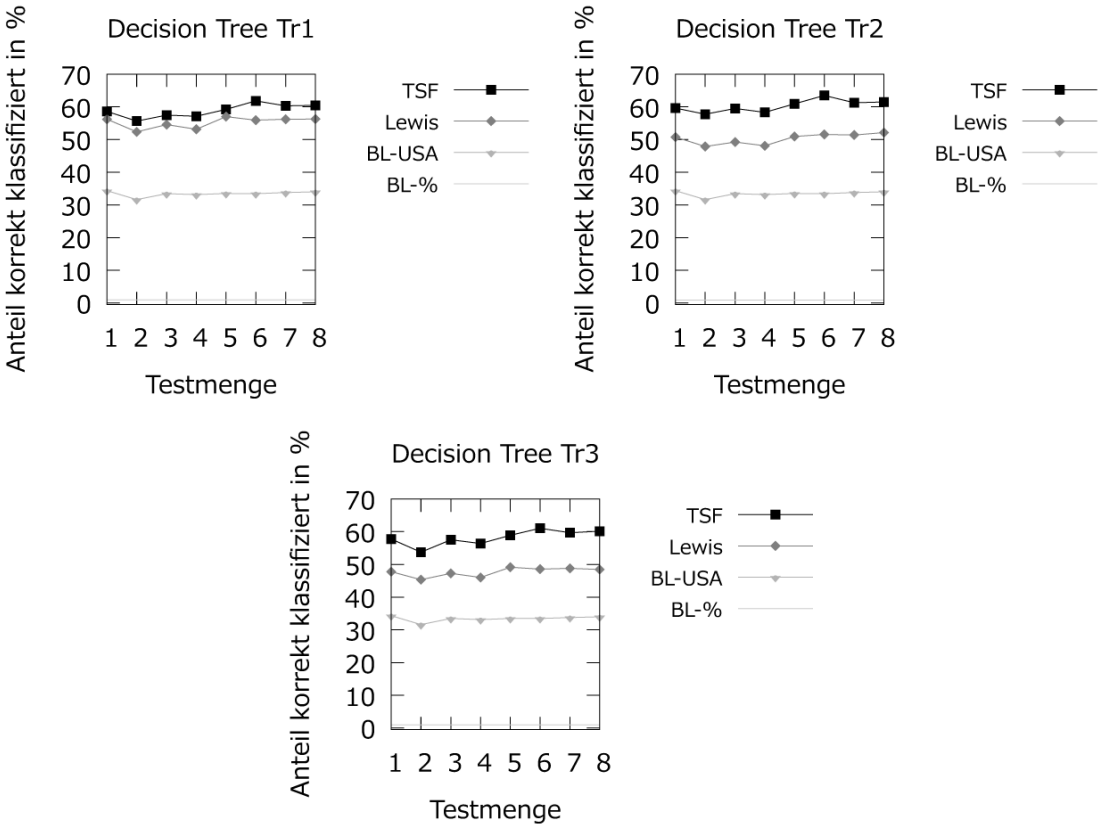


Abbildung 4.67: Ergebnis der Evaluation der Klassifizierung durch den Decision-Tree-Klassifizierer für das Experiment Klass1RV

Auch beim Decision Tree wird die Annahme getroffen, dass die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen sollte als für Te5 bis Te8. Diese Annahme trifft beim Decision Tree für beide Featurevarianten nicht zu.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal um die 60% richtig klassi-

fizierter Dokumente erreicht wird. Für beide Verfahren gilt, dass über alle Trainingsteilmengen wesentlich bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung. Es bestehen Unterschiede beim Abstand zwischen TSF-Feature-Vektoren und einzelwortbasierten Feature-Vektoren in Bezug auf die erreichten Ergebnisse: bei Tr1 ist der Abstand wesentlich geringer als bei den anderen beiden Trainingsteilmengen. Auch hier bestätigt sich, dass die Zusammensetzung der Trainingsteilmenge einen Einfluss auf die Ergebnisqualität hat.

– Ergebnisvergleich k-Nearest-Neighbour
  Bildet man die Ergebnisse, die mit dem 10-Nearest-Neighbour Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Abbildung 4.68 auf S. 328.

  Die abgebildeten Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 36,49 Prozentpunkten zwischen den erreichten Ergebnissen mit den TSF-Features und den Einzelwort-Features.

  Das ist der bisher deutlichste gemessene Abstand in diesem Experiment. Der minimale Abstand betrug 22,85 Prozentpunkte und der maximale Abstand 46,6 Prozentpunkte. Beim 10NN-Klassifizierer liegen die Ergebnisse beider Verfahren wieder weit auseinander.

  Auch beim 10NN wird die Annahme getroffen, dass die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen sollte als für Te5 bis Te8. Diese Annahme trifft beim 10NN für beide Featurevarianten nicht zu.

  Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal um die 70% richtig klassifizierter Dokumente erreicht wird. Für die TSF-Feature-Vektoren gilt, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung. Für die einzelwortbasierten Feature-Vektoren ist das nur bei Tr3 der Fall. Auch hier bestätigt sich, dass die Zusammensetzung der Trainingsteilmenge einen Einfluss auf die Ergebnisqualität hat.

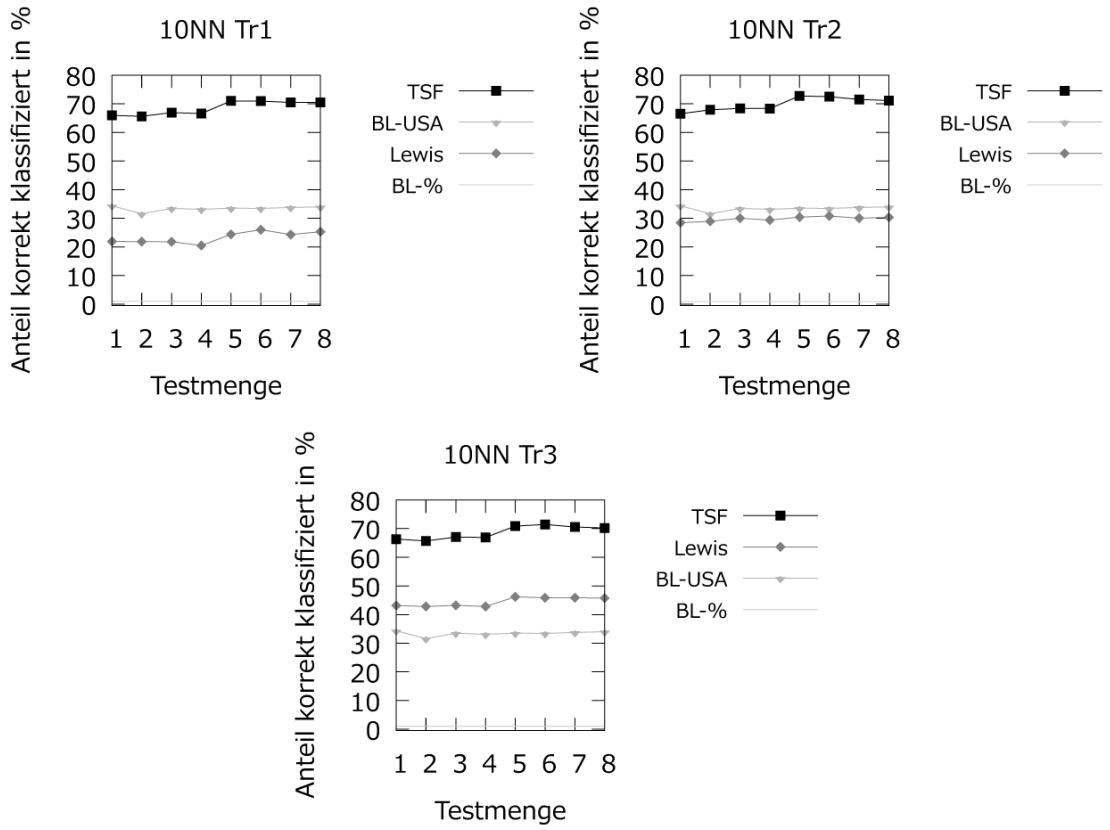


Abbildung 4.68: Ergebnis der Evaluation der Klassifizierung durch den 10-Nearest-Neighbour-Klassifizierer für das Experiment Klass1RV

Bildet man die Ergebnisse, die mit dem 20-Nearest-Neighbour-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Abbildung 4.69 auf S. 329. Die abgebildeten Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 29,38 Prozentpunkten zwischen den erreichten Ergebnissen mit den TSF-Features und den Einzelwort-Features. Der minimale Abstand betrug 21,9 Prozentpunkte und der maximale Abstand 42,2 Prozentpunkte. Beim 20NN-Klassifizierer liegen die Ergebnisse beider Verfahren wieder weit auseinander.

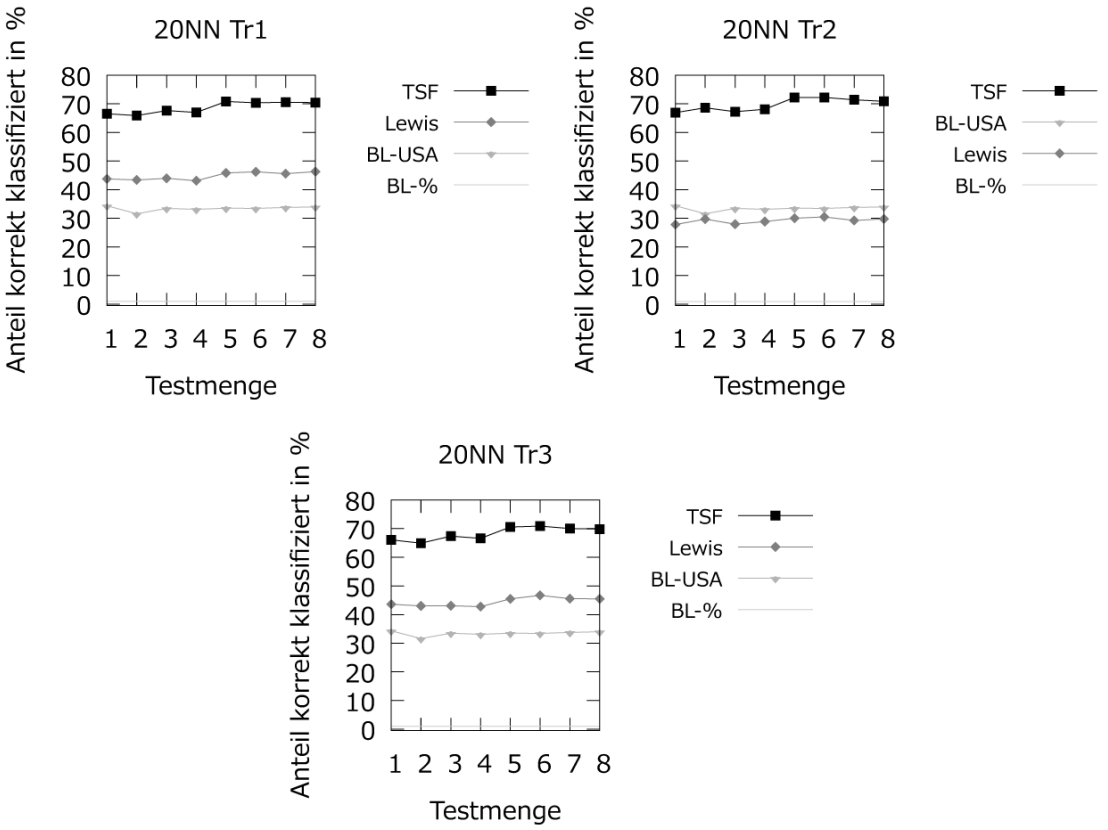


Abbildung 4.69: Ergebnis der Evaluation der Klassifizierung durch den 20-Nearest-Neighbour-Klassifizierer für das Experiment Klass1RV

Auch beim 20NN-Klassifizierer wird die Annahme getroffen, dass die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen sollte als für Te5 bis Te8. Diese Annahme trifft beim 20NN-Klassifizierer für beide Featurevarianten nicht zu.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal um die 70% richtig klassifizierter Dokumente erreicht wird. Für die TSF-Feature-Vektoren gilt, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung. Für die einzelwortbasierten Feature-Vektoren ist das bei Tr2 nicht der Fall. Auch hier bestätigt sich, dass die Zusammensetzung der Trainingsteilmenge einen Einfluss auf die Ergebnisqualität hat.

Bildet man die Ergebnisse, die mit dem 200-Nearest-Neighbour-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Abbildung 4.70.

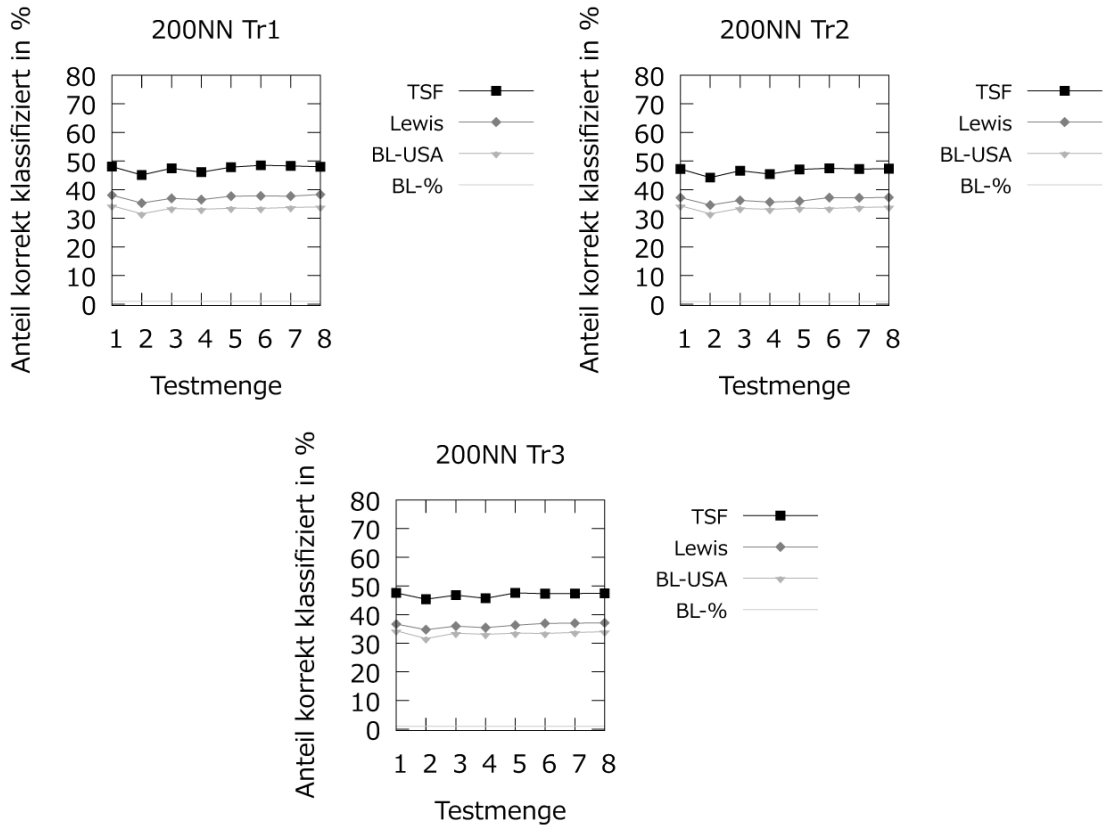


Abbildung 4.70: Ergebnis der Evaluation der Klassifizierung durch den 200-Nearest-Neighbour-Klassifizierer für das Experiment Klass1RV

Auch diese Ergebnisse stützen die der Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 10,28 Prozentpunkten zwischen den erreichten Ergebnissen mit den TSF-Features und den Einzelwort-Features. Der minimale Abstand betrug 9,55 Prozentpunkte und der maximale Abstand 11,25 Prozentpunkte. Beim 200NN-Klassifizierer liegen die Ergebnisse beider Verfahren wieder näher zusammen.

Auch beim 200NN-Klassifizierer wird die Annahme getroffen, dass die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen sollte

als für Te5 bis Te8. Eine klare Beurteilung ist für den 200NN-Klassifizierer aufgrund der sehr gemischten Ergebnisse diesbezüglich nicht möglich. Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal um die 50% richtig klassifizierter Dokumente erreicht wird. Für die TSF-Feature-Vektoren gilt, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung. Für die einzelwortbasierten Feature-Vektoren ist das zwar auch der Fall, jedoch ist der Abstand zwischen den Ergebnissen der besten Baseline und den einzelwortbasierten Feature-Vektoren nicht sehr groß.

Insgesamt ergibt sich für den k-Nearest-Neighbour folgendes Bild:

* Die TSF-Feature-Vektoren erreichen in allen Fällen bessere Ergebnisse als die einzelwortbasierten Feature-Vektoren.
* Die Anzahl der Nachbarn spielt eine entscheidende Rolle bei den erreichten Ergebnissen und dem Abstand zwischen den TSF-Feature-Vektoren und den einzelwortbasierten Feature-Vektoren.
  Die optimale Anzahl von Nachbarn scheint für das durchgeführte Experiment in der Nähe von 10 zu liegen, da die Ergebnisse mit 20 Nachbarn ein wenig schlechter sind und mit 200 Nachbarn deutlich schlechter sind. Das gilt für beide Verfahren. Die richtige Wahl der Anzahl der Nachbarn für den kNN-Algorithmus gehört zu den Problemen dieses Algorithmus, die in Kauf genommen werden müssen, wenn er verwendet werden soll. Die hier gewählten Anzahlen an Nachbarn sind zufällig ausgewählt worden und zeigen sehr deutlich die Abhängigkeit der Qualität der Ergebnisse von dieser Wahl beim Klassifizieren.[1]

– Ergebnisvergleich Support Vector Machine
  Bildet man die Ergebnisse, die mit dem Support-Vector-Machine-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Abbildung 4.71 auf S. 332. Die abgebildeten Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 7,8 Prozentpunkten zwischen den erreichten

1 Zu Möglichkeiten, k festzulegen siehe S. 51 der vorliegenden Arbeit.

Ergebnissen mit den TSF-Features und den Einzelwort-Features. Der minimale Abstand betrug 5,32 Prozentpunkte und der maximale Abstand 10,45 Prozentpunkte. Im Gegensatz zum Naive-Bayes-Algorithmus und zum kNN-Algorithmus liegen die Ergebnisse der Support Vector Machine für beide Verfahren nah beieinander.

Auch bei der SVM wird die Annahme getroffen, dass die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen sollte als für Te5 bis Te8. Diese Annahme trifft jedoch für beide Featurevarianten nicht zu.

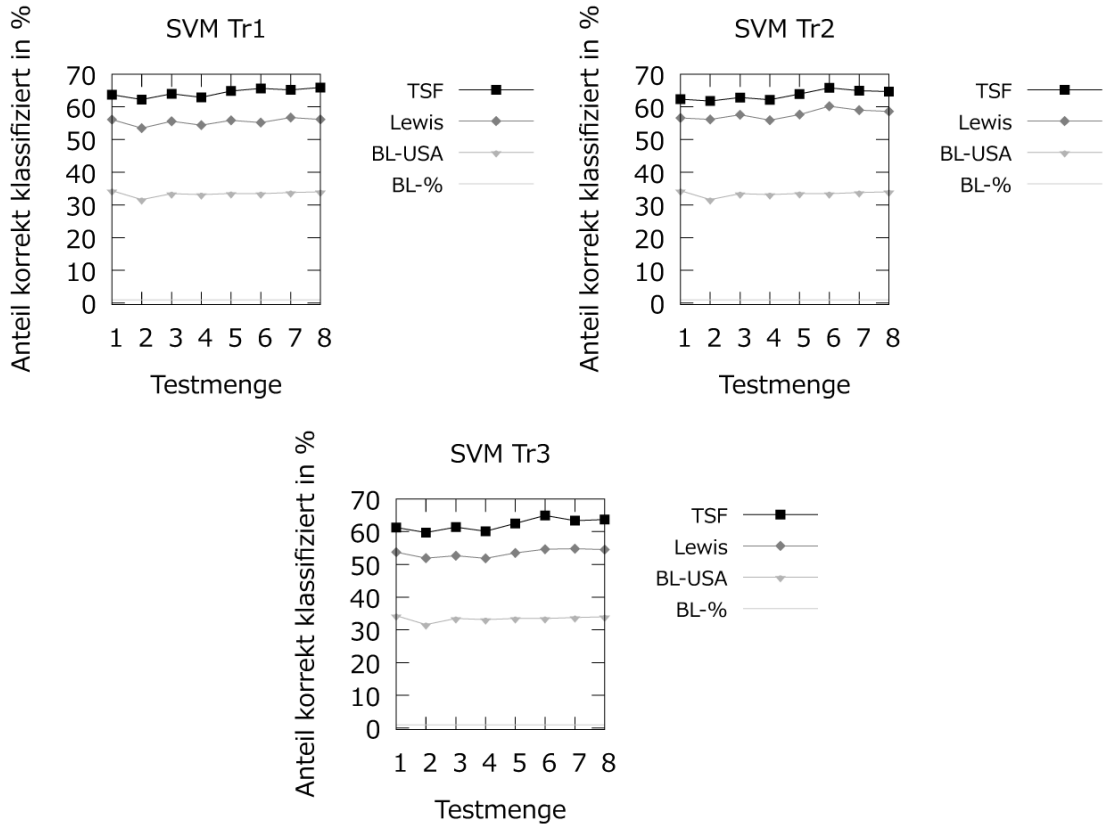


Abbildung 4.71: Ergebnis der Evaluation der Klassifizierung durch den Support-Vector-Machine-Klassifizierer für das Experiment Klass1RV

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal um die 70% richtig klassifizierter Dokumente erreicht wird. Für beide Verfahren gilt, dass über alle

Trainingsteilmengen wesentlich bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung.

– Ergebnisvergleich über alle Algorithmen für das Experiment
Vergleicht man die Ergebnisse des einzelwortbasierten Verfahrens und des Verfahrens mit TSF-Features über alle Algorithmen miteinander, so stellt man Folgendes fest: Das auf TSF-Features basierende Verfahren schneidet beim Klassifizieren von natürlichsprachlichen Dokumenten für alle verwendeten Algorithmen besser ab als das einzelwortbasierte Verfahren.

  Somit stützt dieses erste Experiment die Vermutung, dass wortübergreifende Features qualitativ bessere Ergebnisse beim Klassifizieren von natürlichsprachlichen Dokumenten liefern als Verfahren, die natürlichsprachliche Dokumente in Einzelworte aufteilen und diese als Features für das Klassifizieren benutzen.

  Als Tabelle zusammengefasst ergibt sich für das erste Experiment folgendes Bild[1]:

Tabelle 4.19: Ergebnisse des Experiments Klass1RV

Verfahren	Algorithmus			
	NB	DT	kNN	SVM
TSF-Feature-Vektoren	✓	✓	✓	✓
Lewis-Feature-Vektoren				

Damit erreichen die TSF-Feature-Vektoren in 6 von 6 Fällen ein besseres Klassifizierungergebnis als die einzelwortbasierten Vektoren.[2]

1 Ein Häkchen zeigt dabei das Verfahren an, das für den betreffenden Algorithmus die höchste Qualität über alle Trainings- und Testteilmengen erreicht hat.

2 Hierbei werden die unterschiedlichen Anzahlen an Nachbarn beim kNN-Algorithmus einzeln gezählt.

4.3.2.1.2 Experiment 2

- ID
  Klass1TV
- Bezeichnung
  vektorbasiertes Single-label-Topic-Klassifizieren von Testdaten der Reuters-Daten RCV1-v2
- Trainingsdaten
  - 3 randomisiert zusammengestellte Teilmengen der Trainingsdaten des RCV1-v2 mit jeweils 2.000 Dokumenten: Tr1, Tr2, Tr3
  - Dokumente von Tr1 bis Tr3 gehören zu genau einer Topic-Klasse
- Testdaten
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 2.000 Dokumenten: Te1, Te2
  - Dokumente in Te1 und Te2 haben beliebige Klassenzugehörigkeiten[1]
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 10.000 Dokumenten: Te3, Te4
  - Dokumente in Te3 und Te4 haben beliebige Klassenzugehörigkeiten
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 2.000 Dokumenten: Te5, Te6
  - Dokumente in Te5 und Te6 gehören zu genau einer Topic-Klasse
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 10.000 Dokumenten: Te7, Te8
  - Dokumente in Te7 und Te8 gehören zu genau einer Topic-Klasse
- Ähnlichkeit
  vektorbasiert
- Algorithmen
  - Naive Bayes
  - Decision Tree

1 Zur Erinnerung: Jedes Dokument muss mindestens einer Topic-Klasse zugeordnet sein.

  - k-Nearest-Neighbour mit $k = 10$, 20 und 200 Nachbarn mit Cosinus zur Ähnlichkeitsberechnung zwischen den Vektoren
  - Support Vector Machine SVM^{multiclass} mit Parameter $C = 1,0$ und linearem Kernel

- Baseline
  - randomisierte Zuordnung
  - Majority-Class-Zuordnung zu den zwei Klassen mit den meisten Trainingsdokumenten

- Evaluation
  Anteil der richtig klassifizierten Testdaten pro Testteilmenge an der Anzahl der in der Testteilmenge vorhandenen Dokumente

- Ergebnisse[1]
  Für das Klassifizieren der Testmengen Te1 bis Te8 mit dem Naive-Bayes-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.20, 4.21 und 4.22, zu sehen auf S. 336 bis 337.
  Für das Klassifizieren der Testmengen Te1 bis Te8 mit dem Decision-Tree-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.23, 4.24 und 4.25, zu sehen auf S. 338 bis 339.
  Für das Klassifizieren der Testmengen Te1 bis Te8 mit dem k-Nearest-Neighbour-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3 und 10, 20 und 200 Nachbarn, ergeben sich die Ergebnisse aus den Tabellen 4.26, 4.27, 4.28, 4.29, 4.30, 4.31, 4.32, 4.33 und 4.34, zu sehen auf S. 340 bis 345.
  Für das Klassifizieren der Testmengen Te1 bis Te8 mit dem Support-Vector-Machine-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.35, 4.36 und 4.37, zu sehen auf S. 346 bis 347.

1 In allen Tabellen werden die Ergebnisse auf zwei Stellen nach dem Komma gerundet. Alle Klassifizierer sind im abschließenden Vergleich durch ihre Abkürzungen aufgelistet: NB für Naive Bayes, DT für Decision Tree, kNN für k-Nearest-Neighbour und SVM für Support Vector Machine.

Tabelle 4.20: Ergebnisse des Experiments Klass1TV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1694	306	84,70
	Topic	2	2000	nein	1727	273	86,35
	Topic	3	10000	nein	8422	1578	84,22
	Topic	4	10000	nein	8425	1575	84,25
	Topic	5	2000	ja	1765	235	88,25
	Topic	6	2000	ja	1741	259	87,05
	Topic	7	10000	ja	8752	1248	87,52
	Topic	8	10000	ja	8731	1269	87,31
Lewis-Feature-Vektoren	Topic	1	2000	nein	1462	538	73,10
	Topic	2	2000	nein	1486	514	74,30
	Topic	3	10000	nein	7324	2676	73,24
	Topic	4	10000	nein	7337	2663	73,37
	Topic	5	2000	ja	1542	458	77,10
	Topic	6	2000	ja	1529	471	76,45
	Topic	7	10000	ja	7506	2494	75,06
	Topic	8	10000	ja	7538	2462	75,38

Tabelle 4.21: Ergebnisse des Experiments Klass1TV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1629	371	81,45
	Topic	2	2000	nein	1644	356	82,20
	Topic	3	10000	nein	8134	1866	81,34
	Topic	4	10000	nein	8185	1815	81,85
	Topic	5	2000	ja	1719	281	85,95
	Topic	6	2000	ja	1699	301	84,95
	Topic	7	10000	ja	8475	1525	84,75
	Topic	8	10000	ja	8481	1519	84,81
Lewis-Feature-Vektoren	Topic	1	2000	nein	1464	536	73,20
	Topic	2	2000	nein	1485	515	74,25
	Topic	3	10000	nein	7411	2589	74,11

wird auf der nächsten Seite fortgesetzt

Tabelle 4.21: Ergebnisse des Experiments Klass1TV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr2 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Topic	4	10000	nein	7467	2533	74,67
	Topic	5	2000	ja	1551	449	77,55
	Topic	6	2000	ja	1574	426	78,70
	Topic	7	10000	ja	7707	2293	77,07
	Topic	8	10000	ja	7672	2328	76,72

Tabelle 4.22: Ergebnisse des Experiments Klass1TV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1585	415	79,25
	Topic	2	2000	nein	1610	390	80,50
	Topic	3	10000	nein	7893	2107	78,93
	Topic	4	10000	nein	7867	2133	78,67
	Topic	5	2000	ja	1643	357	82,15
	Topic	6	2000	ja	1647	353	82,35
	Topic	7	10000	ja	8204	1796	82,04
	Topic	8	10000	ja	8208	1792	82,08
Lewis-Feature-Vektoren	Topic	1	2000	nein	1376	624	68,80
	Topic	2	2000	nein	1435	565	71,75
	Topic	3	10000	nein	6899	3101	68,99
	Topic	4	10000	nein	6913	3087	69,13
	Topic	5	2000	ja	1448	552	72,40
	Topic	6	2000	ja	1402	598	70,10
	Topic	7	10000	ja	7112	2888	71,12
	Topic	8	10000	ja	7083	2917	70,83

Tabelle 4.23: Ergebnisse des Experiments Klass1TV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1524	476	76,20
	Topic	2	2000	nein	1520	480	76,00
	Topic	3	10000	nein	7579	2421	75,79
	Topic	4	10000	nein	7497	2503	74,97
	Topic	5	2000	ja	1516	484	75,80
	Topic	6	2000	ja	1541	459	77,05
	Topic	7	10000	ja	7743	2257	77,43
	Topic	8	10000	ja	7761	2239	77,61
Lewis-Feature-Vektoren	Topic	1	2000	nein	1512	488	75,60
	Topic	2	2000	nein	1560	440	78,00
	Topic	3	10000	nein	7792	2208	77,92
	Topic	4	10000	nein	7855	2145	78,55
	Topic	5	2000	ja	1635	365	81,75
	Topic	6	2000	ja	1616	384	80,80
	Topic	7	10000	ja	8019	1981	80,19
	Topic	8	10000	ja	8065	1935	80,65

Tabelle 4.24: Ergebnisse des Experiments Klass1TV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1499	501	74,95
	Topic	2	2000	nein	1462	538	73,10
	Topic	3	10000	nein	7466	2534	74,66
	Topic	4	10000	nein	7377	2623	73,77
	Topic	5	2000	ja	1530	470	76,50
	Topic	6	2000	ja	1526	474	76,30
	Topic	7	10000	ja	7509	2491	75,09
	Topic	8	10000	ja	7681	2319	76,81
Lewis-Feature-Vektoren	Topic	1	2000	nein	1579	421	78,95
	Topic	2	2000	nein	1538	462	76,90
	Topic	3	10000	nein	7836	2164	78,36

wird auf der nächsten Seite fortgesetzt

Tabelle 4.24: Ergebnisse des Experiments Klass1TV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr2 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Topic	4	10000	nein	7758	2242	77,58
	Topic	5	2000	ja	1664	336	83,20
	Topic	6	2000	ja	1619	381	80,95
	Topic	7	10000	ja	7909	2091	79,09
	Topic	8	10000	ja	7991	2009	79,91

Tabelle 4.25: Ergebnisse des Experiments Klass1TV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1535	465	76,75
	Topic	2	2000	nein	1515	485	75,75
	Topic	3	10000	nein	7475	2525	74,75
	Topic	4	10000	nein	7491	2509	74,91
	Topic	5	2000	ja	1542	458	77,10
	Topic	6	2000	ja	1556	444	77,80
	Topic	7	10000	ja	7547	2453	75,47
	Topic	8	10000	ja	7670	2330	76,70
Lewis-Feature-Vektoren	Topic	1	2000	nein	1523	477	76,15
	Topic	2	2000	nein	1529	471	76,45
	Topic	3	10000	nein	7720	2280	77,20
	Topic	4	10000	nein	7711	2289	77,11
	Topic	5	2000	ja	1621	379	81,05
	Topic	6	2000	ja	1567	433	78,35
	Topic	7	10000	ja	7971	2029	79,71
	Topic	8	10000	ja	8009	1991	80,09

Tabelle 4.26: Ergebnisse des Experiments Klass1TV für den 10NN-Klasifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1784	216	89,20
	Topic	2	2000	nein	1791	209	89,55
	Topic	3	10000	nein	8885	1115	88,85
	Topic	4	10000	nein	9014	986	90,14
	Topic	5	2000	ja	1843	157	92,15
	Topic	6	2000	ja	1846	154	92,30
	Topic	7	10000	ja	9212	788	92,12
	Topic	8	10000	ja	9172	828	91,72
Lewis-Feature-Vektoren	Topic	1	2000	nein	1029	971	51,45
	Topic	2	2000	nein	1046	954	52,30
	Topic	3	10000	nein	5120	4880	51,20
	Topic	4	10000	nein	5189	4811	51,89
	Topic	5	2000	ja	932	1068	46,60
	Topic	6	2000	ja	995	1005	49,75
	Topic	7	10000	ja	4760	5240	47,60
	Topic	8	10000	ja	4796	5204	47,96

Tabelle 4.27: Ergebnisse des Experiments Klass1TV für den 20NN-Klasifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1726	274	86,30
	Topic	2	2000	nein	1741	259	87,05
	Topic	3	10000	nein	8707	1293	87,07
	Topic	4	10000	nein	8762	1238	87,62
	Topic	5	2000	ja	1818	182	90,90
	Topic	6	2000	ja	1821	179	91,05
	Topic	7	10000	ja	9029	971	90,29
	Topic	8	10000	ja	9050	950	90,50
Lewis-Feature-Vektoren	Topic	1	2000	nein	1023	977	51,15
	Topic	2	2000	nein	1043	957	52,15
	Topic	3	10000	nein	5103	4897	51,03

wird auf der nächsten Seite fortgesetzt

Tabelle 4.27: Ergebnisse des Experiments Klass1TV für den 20NN-Klasifizierer für Trainingsmenge Tr1 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Topic	4	10000	nein	5190	4810	51,90
	Topic	5	2000	ja	933	1067	46,65
	Topic	6	2000	ja	985	1015	49,25
	Topic	7	10000	ja	4731	5269	47,31
	Topic	8	10000	ja	4722	5278	47,22

Tabelle 4.28: Ergebnisse des Experiments Klass1TV für den 200NN-Klasifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1622	378	81,10
	Topic	2	2000	nein	1639	361	81,95
	Topic	3	10000	nein	8170	1830	81,70
	Topic	4	10000	nein	8158	1842	81,58
	Topic	5	2000	ja	1733	267	86,65
	Topic	6	2000	ja	1700	300	85,00
	Topic	7	10000	ja	8480	1520	84,80
	Topic	8	10000	ja	8456	1544	84,56
Lewis-Feature-Vektoren	Topic	1	2000	nein	1284	716	64,20
	Topic	2	2000	nein	1215	785	60,75
	Topic	3	10000	nein	6338	3662	63,38
	Topic	4	10000	nein	6159	3841	61,59
	Topic	5	2000	ja	1220	780	61,00
	Topic	6	2000	ja	1158	842	57,90
	Topic	7	10000	ja	6125	3875	61,25
	Topic	8	10000	ja	6005	3995	60,05

Tabelle 4.29: Ergebnisse des Experiments Klass1TV für den 10NN-Klasifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1820	180	91,00
	Topic	2	2000	nein	1832	168	91,60
	Topic	3	10000	nein	9095	905	90,95
	Topic	4	10000	nein	9170	830	91,70
	Topic	5	2000	ja	1884	116	94,20
	Topic	6	2000	ja	1885	115	94,25
	Topic	7	10000	ja	9401	599	94,01
	Topic	8	10000	ja	9327	673	93,27
Lewis-Feature-Vektoren	Topic	1	2000	nein	1035	965	51,75
	Topic	2	2000	nein	1041	959	52,05
	Topic	3	10000	nein	5227	4773	52,27
	Topic	4	10000	nein	5345	4655	53,45
	Topic	5	2000	ja	972	1028	48,60
	Topic	6	2000	ja	1009	991	50,45
	Topic	7	10000	ja	4924	5076	49,24
	Topic	8	10000	ja	4851	5149	48,51

Tabelle 4.30: Ergebnisse des Experiments Klass1TV für den 20NN-Klasifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1801	199	90,05
	Topic	2	2000	nein	1801	199	90,05
	Topic	3	10000	nein	8969	1031	89,69
	Topic	4	10000	nein	9045	955	90,45
	Topic	5	2000	ja	1847	153	92,35
	Topic	6	2000	ja	1873	127	93,65
	Topic	7	10000	ja	9274	726	92,74
	Topic	8	10000	ja	9222	778	92,22
Lewis-Feature-Vektoren	Topic	1	2000	nein	1032	968	51,60
	Topic	2	2000	nein	1047	953	52,35
	Topic	3	10000	nein	5213	4787	52,13

wird auf der nächsten Seite fortgesetzt

Tabelle 4.30: Ergebnisse des Experiments Klass1TV für den 20NN-Klasifizierer für Trainingsmenge Tr2 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Topic	4	10000	nein	5322	4678	53,22
	Topic	5	2000	ja	961	1039	48,05
	Topic	6	2000	ja	1005	995	50,25
	Topic	7	10000	ja	4898	5102	48,98
	Topic	8	10000	ja	4820	5180	48,20

Tabelle 4.31: Ergebnisse des Experiments Klass1TV für den 200NN-Klasifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1634	366	81,70
	Topic	2	2000	nein	1655	345	82,75
	Topic	3	10000	nein	8297	1703	82,97
	Topic	4	10000	nein	8237	1763	82,37
	Topic	5	2000	ja	1740	260	87,00
	Topic	6	2000	ja	1721	279	86,05
	Topic	7	10000	ja	8581	1419	85,81
	Topic	8	10000	ja	8559	1441	85,59
Lewis-Feature-Vektoren	Topic	1	2000	nein	1294	706	64,70
	Topic	2	2000	nein	1240	760	62,00
	Topic	3	10000	nein	6396	3604	63,96
	Topic	4	10000	nein	6258	3742	62,58
	Topic	5	2000	ja	1245	755	62,25
	Topic	6	2000	ja	1185	815	59,25
	Topic	7	10000	ja	6243	3757	62,43
	Topic	8	10000	ja	6044	3956	60,44

Tabelle 4.32: Ergebnisse des Experiments Klass1TV für den 10NN-Klasifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1750	250	87,50
	Topic	2	2000	nein	1766	234	88,30
	Topic	3	10000	nein	8745	1255	87,45
	Topic	4	10000	nein	8869	1131	88,69
	Topic	5	2000	ja	1806	194	90,30
	Topic	6	2000	ja	1838	162	91,90
	Topic	7	10000	ja	9059	941	90,59
	Topic	8	10000	ja	9036	964	90,36
Lewis-Feature-Vektoren	Topic	1	2000	nein	1058	942	52,90
	Topic	2	2000	nein	1062	938	53,10
	Topic	3	10000	nein	5155	4845	51,55
	Topic	4	10000	nein	5256	4744	52,56
	Topic	5	2000	ja	964	1036	48,20
	Topic	6	2000	ja	997	1003	49,85
	Topic	7	10000	ja	4874	5126	48,74
	Topic	8	10000	ja	4856	5144	48,56

Tabelle 4.33: Ergebnisse des Experiments Klass1TV für den 20NN-Klasifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1723	277	86,15
	Topic	2	2000	nein	1729	271	86,45
	Topic	3	10000	nein	8524	1476	85,24
	Topic	4	10000	nein	8600	1400	86,00
	Topic	5	2000	ja	1767	233	88,35
	Topic	6	2000	ja	1795	205	89,75
	Topic	7	10000	ja	8895	1105	88,95
	Topic	8	10000	ja	8866	1134	88,66
Lewis-Feature-Vektoren	Topic	1	2000	nein	1054	946	52,70
	Topic	2	2000	nein	1067	933	53,35
	Topic	3	10000	nein	5129	4871	51,29

wird auf der nächsten Seite fortgesetzt

Tabelle 4.33: Ergebnisse des Experiments Klass1TV für den 20NN-Klasifizierer für Trainingsmenge Tr3 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Topic	4	10000	nein	5231	4769	52,31
	Topic	5	2000	ja	964	1036	48,20
	Topic	6	2000	ja	993	1007	49,65
	Topic	7	10000	ja	4849	5151	48,49
	Topic	8	10000	ja	4833	5167	48,33

Tabelle 4.34: Ergebnisse des Experiments Klass1TV für den 200NN-Klasifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1536	464	76,80
	Topic	2	2000	nein	1558	442	77,90
	Topic	3	10000	nein	7706	2294	77,06
	Topic	4	10000	nein	7623	2377	76,23
	Topic	5	2000	ja	1627	373	81,35
	Topic	6	2000	ja	1606	394	80,30
	Topic	7	10000	ja	7981	2019	79,81
	Topic	8	10000	ja	7957	2043	79,57
Lewis-Feature-Vektoren	Topic	1	2000	nein	1306	694	65,30
	Topic	2	2000	nein	1230	770	61,50
	Topic	3	10000	nein	6348	3652	63,48
	Topic	4	10000	nein	6187	3813	61,87
	Topic	5	2000	ja	1249	751	62,45
	Topic	6	2000	ja	1154	846	57,70
	Topic	7	10000	ja	6174	3826	61,74
	Topic	8	10000	ja	6086	3914	60,86

Tabelle 4.35: Ergebnisse des Experiments Klass1TV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1814	186	90,70
	Topic	2	2000	nein	1787	213	89,35
	Topic	3	10000	nein	9040	960	90,40
	Topic	4	10000	nein	9128	872	91,28
	Topic	5	2000	ja	1859	141	92,95
	Topic	6	2000	ja	1882	118	94,10
	Topic	7	10000	ja	9329	671	93,29
	Topic	8	10000	ja	9274	726	92,74
Lewis-Feature-Vektoren	Topic	1	2000	nein	1635	365	81,75
	Topic	2	2000	nein	1670	330	83,50
	Topic	3	10000	nein	8252	1748	82,52
	Topic	4	10000	nein	8280	1720	82,80
	Topic	5	2000	ja	1698	302	84,90
	Topic	6	2000	ja	1705	295	85,25
	Topic	7	10000	ja	8544	1456	85,44
	Topic	8	10000	ja	8384	1616	83,84

Tabelle 4.36: Ergebnisse des Experiments Klass1TV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1852	148	92,60
	Topic	2	2000	nein	1822	178	91,10
	Topic	3	10000	nein	9010	990	90,10
	Topic	4	10000	nein	9112	888	91,12
	Topic	5	2000	ja	1861	139	93,05
	Topic	6	2000	ja	1853	147	92,65
	Topic	7	10000	ja	9296	704	92,96
	Topic	8	10000	ja	9246	754	92,46
Lewis-Feature-Vektoren	Topic	1	2000	nein	1715	285	85,75
	Topic	2	2000	nein	1690	310	84,50
	Topic	3	10000	nein	8575	1425	85,75

wird auf der nächsten Seite fortgesetzt

Tabelle 4.36: Ergebnisse des Experiments Klass1TV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr2 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Topic	4	10000	nein	8533	1467	85,33
	Topic	5	2000	ja	1770	230	88,50
	Topic	6	2000	ja	1753	247	87,65
	Topic	7	10000	ja	8830	1170	88,30
	Topic	8	10000	ja	8789	1211	87,89

Tabelle 4.37: Ergebnisse des Experiments Klass1TV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Topic	1	2000	nein	1774	226	88,70
	Topic	2	2000	nein	1734	266	86,70
	Topic	3	10000	nein	8761	1239	87,61
	Topic	4	10000	nein	8807	1193	88,07
	Topic	5	2000	ja	1806	194	90,30
	Topic	6	2000	ja	1814	186	90,70
	Topic	7	10000	ja	9060	940	90,60
	Topic	8	10000	ja	9018	982	90,18
Lewis-Feature-Vektoren	Topic	1	2000	nein	1743	257	87,15
	Topic	2	2000	nein	1701	299	85,05
	Topic	3	10000	nein	8619	1381	86,19
	Topic	4	10000	nein	8727	1273	87,27
	Topic	5	2000	ja	1776	224	88,80
	Topic	6	2000	ja	1783	217	89,15
	Topic	7	10000	ja	8857	1143	88,57
	Topic	8	10000	ja	8855	1145	88,55

- Interpretation der Ergebnisse
  - Einführung
    Im Folgenden werden die vorgestellten Ergebnisse für die auf den Suffix Arrays basierenden Vektoren und für die Lewis-Feature-Vektoren einzeln pro Algorithmus miteinander verglichen. Abschließend wird ein allgemeiner Ver-

gleich zwischen den zwei Verfahren bezogen auf das gesamte Experiment über alle Algorithmen und Trainingsteilmengen gezogen.

– Ergebnisvergleich Naive Bayes
Bildet man die Ergebnisse, die mit dem Naive-Bayes-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Folgendes[1]:

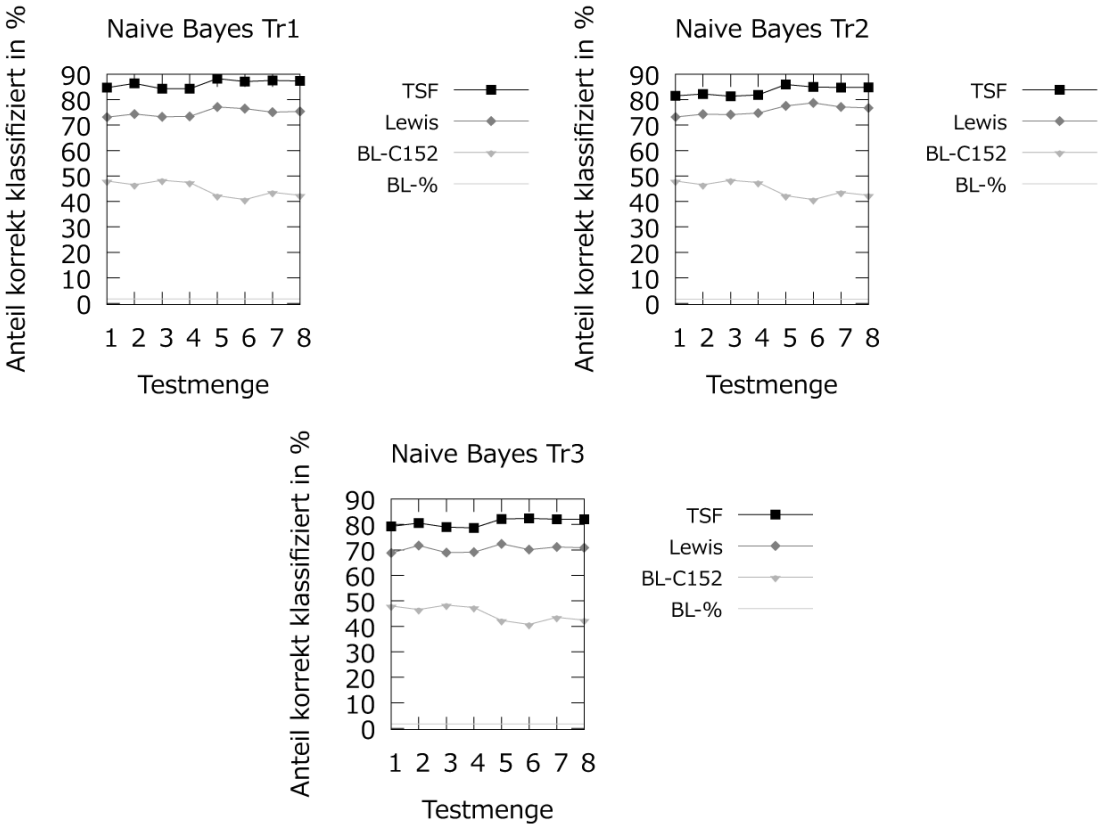


Abbildung 4.72: Ergebnis der Evaluation der Klassifizierung durch den Naive-Bayes-Klassifizierer für das Experiment Klass1TV

Der erste Klassifizierer dieses Experiments erreicht für die wortübergreifenden Features bessere Klassifizierungsergebnisse als für einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 9,82 Prozentpunkten zwischen den erreich-

1 Die Abkürzungen in den Diagrammen bedeuten für dieses Experiment Folgendes: Baseline wird mit „BL“ abgekürzt, „C152“ steht für den Klassennamen der Klasse mit den meisten Trainingsdokumenten und „%“ steht für die Baseline mit der randomisierten Zuordnung.

ten Ergebnissen mit den TSF-Features und den Einzelwort-Features. Der minimale Abstand betrug 6,25 Prozentpunkte und der maximale Abstand 12,46 Prozentpunkte.

Die Trainingsteilmengen bestehen alle drei aus Dokumenten, die nur eine Klassenzuordnung der betrachteten Klassenfamilie Topic haben. Um Unterschiede feststellen zu können, wurden in den Testteilmengen zwei verschiedene Auswahlkriterien bezüglich der Klassenzuordnungen der Dokumente getroffen. So ist der erste Teil dieser Teilmengen, die Te1 bis Te4, keiner Einschränkung unterworfen hinsichtlich der Anzahl der Klassenzuordnungen der betrachteten Klassenfamilie, die die enthaltenen Dokumente haben. Das bedeutet, die Dokumente haben mindestens eine Topic-Zuordnung, können aber auch mehrere haben. Der zweite Teil dieser Teilmengen dagegen enthält nur Testdokumente, die *genau eine* Topic-Zuordnung haben. Die Annahme ist, dass, wenn mehr als eine Klasse korrekt sein kann, eine höhere Wahrscheinlichkeit besteht, dass eine der korrekten Klassen zugeordnet wurde.[1] Also sollte die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen als für Te5 bis Te8. Bei beiden Verfahren trifft diese Annahme für die Klassenfamilie Topic für den Naive-Bayes-Algorithmus nicht zu.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal um die 85% richtig klassifizierter Dokumente erreicht wird. Für beide Verfahren gilt, dass über alle Trainingsteilmengen wesentlich bessere Ergebnisse erzielt werden als mit einer zufälligen Zuordnung der Testdokumente zu den zuordenbaren Klassen, d.h., es werden bessere Ergebnisse erreicht als mit der Baseline-Klassifizierung.

– Ergebnisvergleich Decision Tree
  Bildet man die Ergebnisse, die mit dem Decision-Tree-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man die Abbildung 4.73 auf S. 350. Die Ergebnisse des Decision Trees liegen für beide Verfahren nah beieinander. So beträgt der kleinste Abstand zwischen beiden Verfahren 0,6 Prozentpunkte. Als maximaler Abstand für den Decision Tree ergibt sich ein Wert von 6,7 Prozentpunkten. Für die Trai-

1 Zusätzlich ist bei dieser Klassenfamilie zu beachten, dass aufgrund der Hierarchie auch die übergeordneten Klassen sowohl der zugeordneten Klasse als auch der korrekte(n) Klasse(n) einbezogen werden.

ningsteilmengen Tr1 und Tr2 ist das einzelwortbasierte Verfahren besser, bei Tr3 erreichen beide Verfahren fast das gleiche Ergebnis. Im Durchschnitt bedeutet das, dass das einzelwortbasierte Verfahren um 3,05 Prozentpunkte besser abschneidet als das Verfahren mit TSF-Features. Dieser Abstand ist zwar nicht so deutlich, aber er ist vorhanden. Es handelt sich also um das erste Experiment, in dem für einen Algorithmus das einzelwortbasierte Verfahren bessere Ergebnisse erzielt als das Verfahren, welches auf TSF-Features basiert.

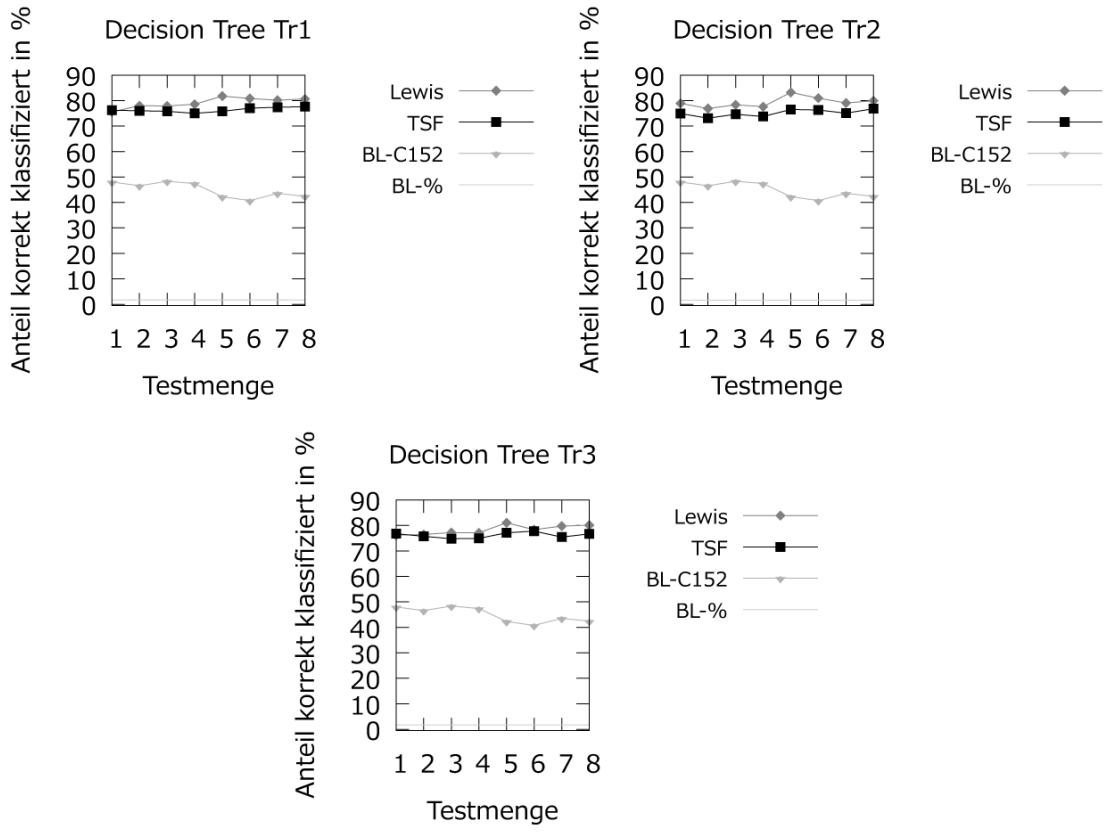


Abbildung 4.73: Ergebnis der Evaluation der Klassifizierung durch den Decision-Tree-Klassifizierer für das Experiment Klass1TV

Eine mögliche Erklärung dafür ist: *overfitting*. Gerade die TSF-Features sind, anders als die einzelwortbasierten, aus denen Stoppworte entfernt und die gestemmt wurden, sehr nah am eigentlichen Dokumentinhalt. Da der Decision Tree darauf basiert, nicht alle Features beim Klassifizieren in Betracht zu ziehen, sondern nur die Auswahl, die die Trainingsdaten korrekt klassifiziert, kann es sein, dass beim Klassifizieren der Testdaten genau diese Features

diese Daten nicht gut klassifizieren können, weil sie zu sehr auf die Trainingsdaten zugeschnitten sind. Das müsste in einer anschließenden Arbeit überprüft werden, indem Methoden angewendet werden, dieses *overfitting* zu reduzieren.[1]

Auch beim Decision Tree wird die Annahme getroffen, dass die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen sollte als für Te5 bis Te8. Diese Annahme trifft beim Decision Tree für beide Featurevarianten nicht zu.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal um die 80% richtig klassifizierter Dokumente erreicht wird. Für beide Verfahren gilt, dass über alle Trainingsteilmengen wesentlich bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung. Aufgrund des besseren Abschneidens der einzelwortbasierten Feature-Vektoren in 2 von 3 Fällen wird insgesamt für den Decision-Tree-Klassifizierer für die Topic-Klassenfamilie ein besseres Abschneiden des einzelwortbasierten Verfahrens über alle drei Trainingsteilmengen gewertet.

– Ergebnisvergleich k-Nearest-Neighbour
  Bildet man die Ergebnisse, die mit dem 10-Nearest-Neighbour-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Abbildung 4.74 auf S. 352.

  Die abgebildeten Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 40,48 Prozentpunkten zwischen den erreichten Ergebnissen mit den TSF-Features und den Einzelwort-Features. Das ist der bisher deutlichste gemessene Abstand in diesem Experiment.

  Der minimale Abstand betrug 34,6 Prozentpunkte und der maximale Abstand 45,6 Prozentpunkte. Beim 10NN-Klassifizierer liegen die Ergebnisse beider Verfahren wieder weit auseinander.

1 Die Überprüfung ist aufwändig, da laut Hawkins (2004), S. 4, immer ein *Vergleich* mit einem einfacheren Decision Tree dazugehören würde, um festzustellen, ob es sich bei dem hier verwendeten um einen handelt, der zu sehr auf die Trainingsdaten zugeschnitten ist.

Auch beim 10NN-Klassifizierer wird die Annahme getroffen, dass die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen sollte als für Te5 bis Te8. Diese Annahme trifft beim 10NN-Klassifizierer für die TSF-Feature-Vektoren nicht zu. Bei den einzelwortbasierten Feature-Vektoren trifft sie zu.

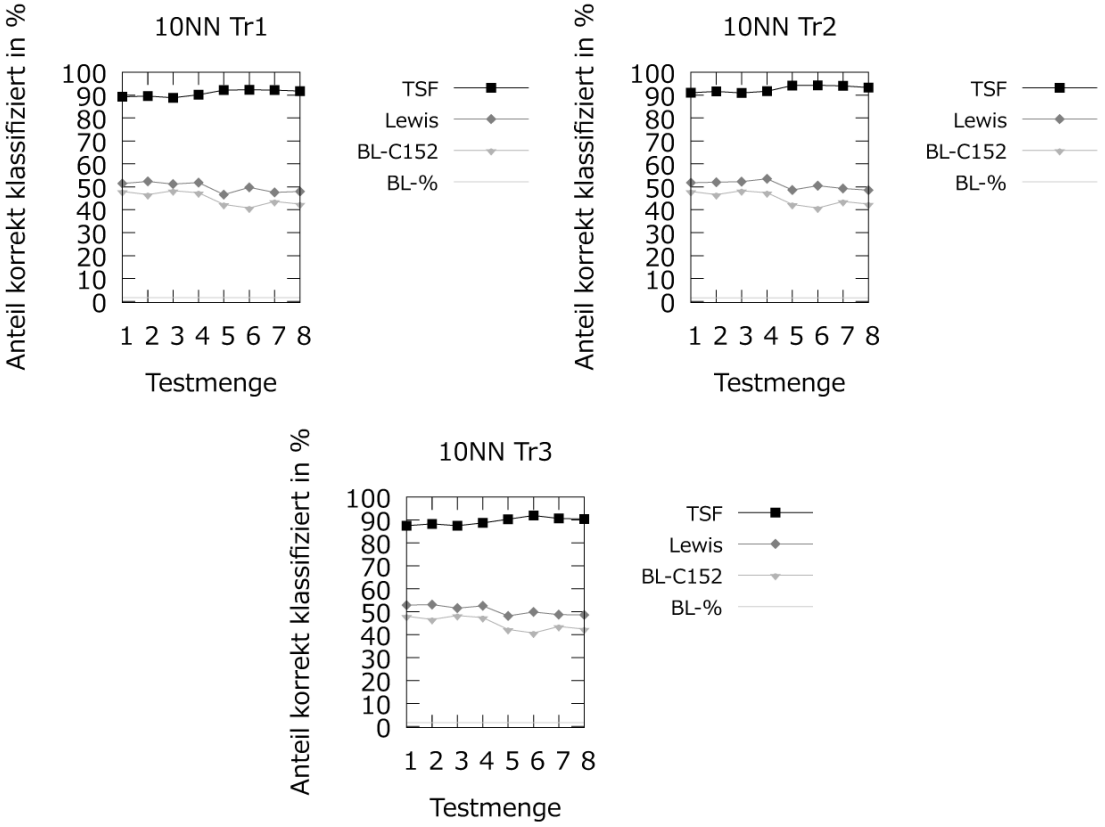


Abbildung 4.74: Ergebnis der Evaluation der Klassifizierung durch den 10-Nearest-Neighbour-Klassifizierer für das Experiment Klass1TV

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von um die 90% richtig klassifizierter Dokumente erreicht wird. Für beide Verfahren gilt, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung.

Bildet man die Ergebnisse, die mit dem 20-Nearest-Neighbour Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Folgendes:

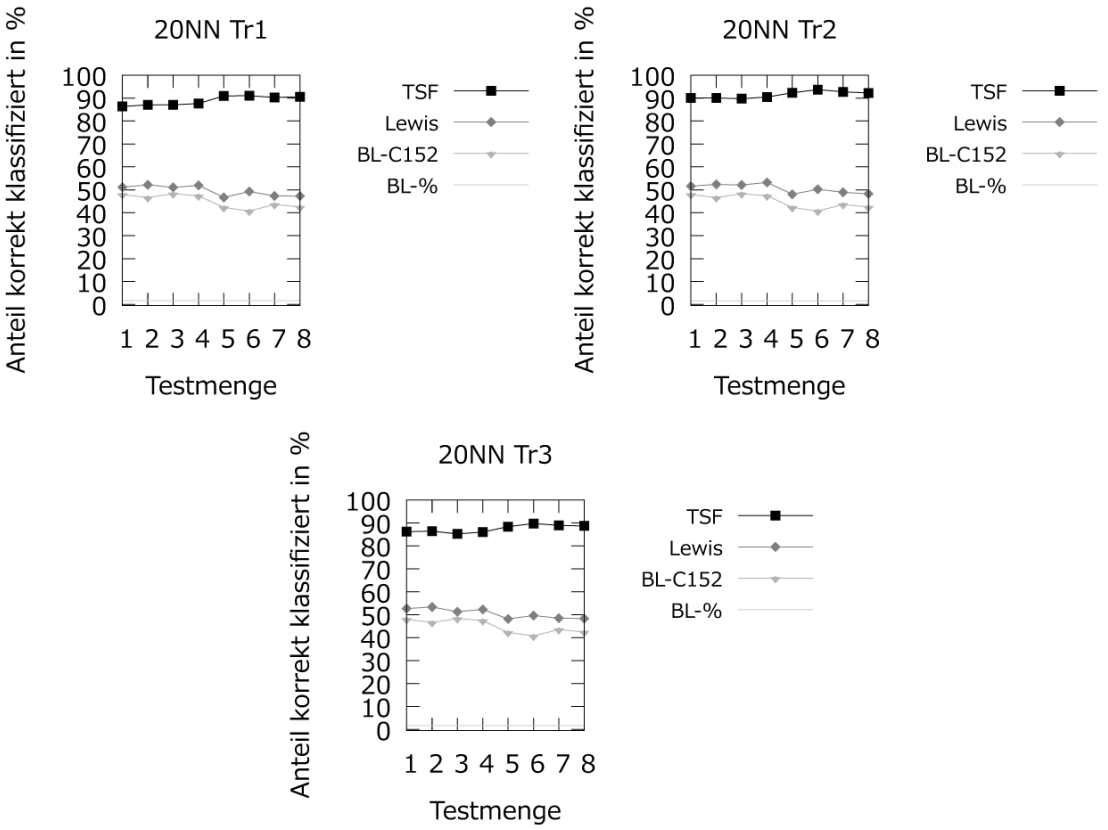


Abbildung 4.75: Ergebnis der Evaluation der Klassifizierung durch den 20-Nearest-Neighbour-Klassifizierer für das Experiment Klass1TV

Die abgebildeten Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 38,99 Prozentpunkten zwischen den erreichten Ergebnissen mit den TSF-Features und den Einzelwort-Features. Der minimale Abstand betrug 33,1 Prozentpunkte und der maximale Abstand 44,3 Prozentpunkte. Beim 20NN-Klassifizierer liegen die Ergebnisse beider Verfahren weit auseinander.

Auch beim 20NN-Klassifizierer wird die Annahme getroffen, dass die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen sollte als für

Te5 bis Te8. Diese Annahme trifft beim 20NN-Klassifizierer für die TSF-Feature-Vektoren nicht zu. Bei den einzelwortbasierten Feature-Vektoren trifft sie zu.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von um die 90% richtig klassifizierter Dokumente erreicht wird. Für beide gilt, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung.

Bildet man die Ergebnisse, die mit dem 200-Nearest-Neighbour-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Folgendes:

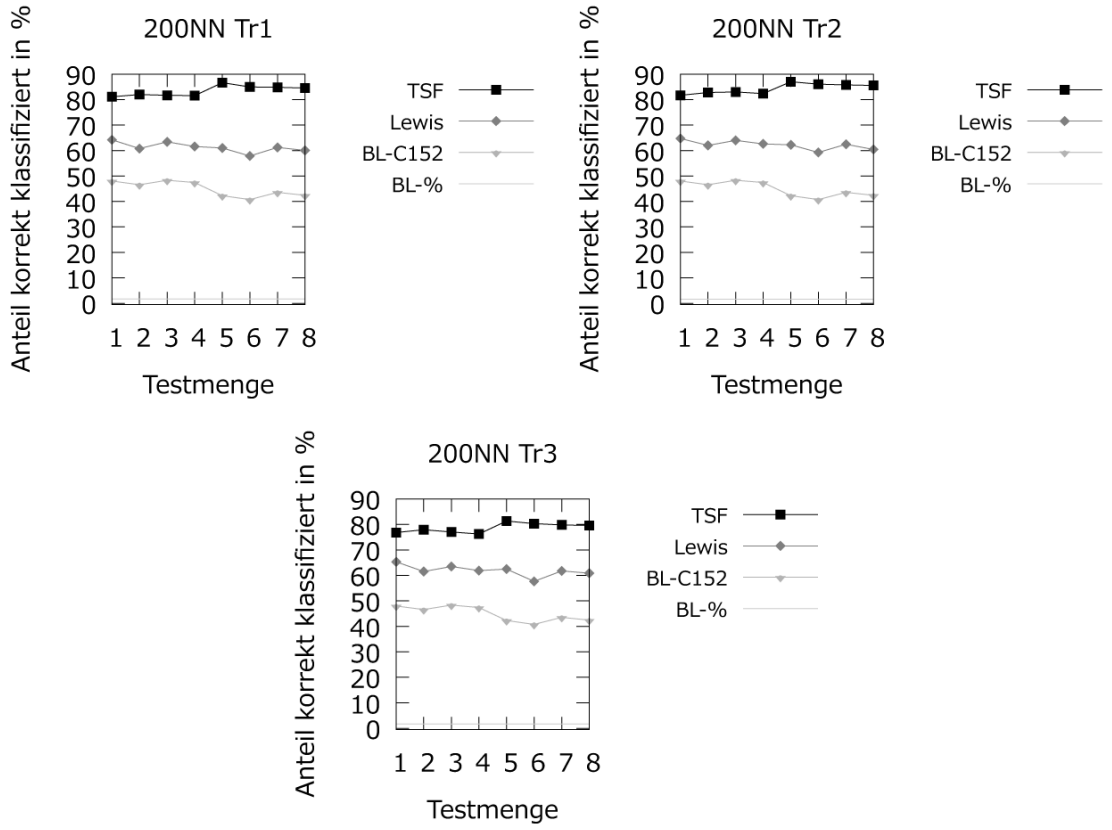


Abbildung 4.76: Ergebnis der Evaluation der Klassifizierung durch den 200-Nearest-Neighbour-Klassifizierer für das Experiment Klass1TV

Auch diese Ergebnisse stützen die der Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als

einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 20,33 Prozentpunkten zwischen den erreichten Ergebnissen mit den TSF-Features und den Einzelwort-Features. Der minimale Abstand betrug 11,5 Prozentpunkte und der maximale Abstand 27,1 Prozentpunkte.

Auch beim 200NN-Klassifizierer wird die Annahme getroffen, dass die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen sollte als für Te5 bis Te8. Diese Annahme trifft beim 200NN-Klassifizierer für die TSF-Feature-Vektoren nicht zu. Bei den einzelwortbasierten Feature-Vektoren trifft sie zu.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von um die 85% richtig klassifizierter Dokumente erreicht wird. Für beide Verfahren gilt, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung.

Insgesamt ergibt sich für den k-Nearest-Neighbour-Klassifizierer folgendes Bild:

* Die TSF-Feature-Vektoren erreichen in allen Fällen bessere Ergebnisse als die einzelwortbasierten Feature-Vektoren.
* Die Anzahl der Nachbarn spielt eine entscheidende Rolle bei den erreichten Ergebnissen und dem Abstand zwischen den TSF-Feature-Vektoren und den einzelwortbasierten Feature-Vektoren.
  Die optimale Anzahl von Nachbarn scheint für das durchgeführte Experiment in der Nähe von 10 zu liegen, da die Ergebnisse mit 20 Nachbarn ein wenig schlechter sind und mit 200 Nachbarn noch schlechter sind. Das gilt für beide Verfahren.

– Ergebnisvergleich Support Vector Machine
  Bildet man die Ergebnisse, die mit dem Support-Vector-Machine-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Abbildung 4.77 auf S. 356. Die Ergebnisse der Support Vector Machine liegen für beide Verfahren nah beieinander. So beträgt der kleinste Abstand zwischen beiden Verfahren 0,8 Prozentpunkte. Als maximaler Abstand für die Support Vector Machine ergibt sich ein Wert von

8,95 Prozentpunkten. Im Durchschnitt bedeutet das, dass die TSF-Feature-Vektoren um 4,97 Prozentpunkte besser abschneiden als die einzelwortbasierten Feature-Vektoren. Dieser Abstand ist zwar nicht so deutlich, aber er ist vorhanden. Das bedeutet, die TSF-Feature-Vektoren schneiden für alle Trainingsteilmengen besser ab als die einzelwortbasierten Feature-Vektoren.

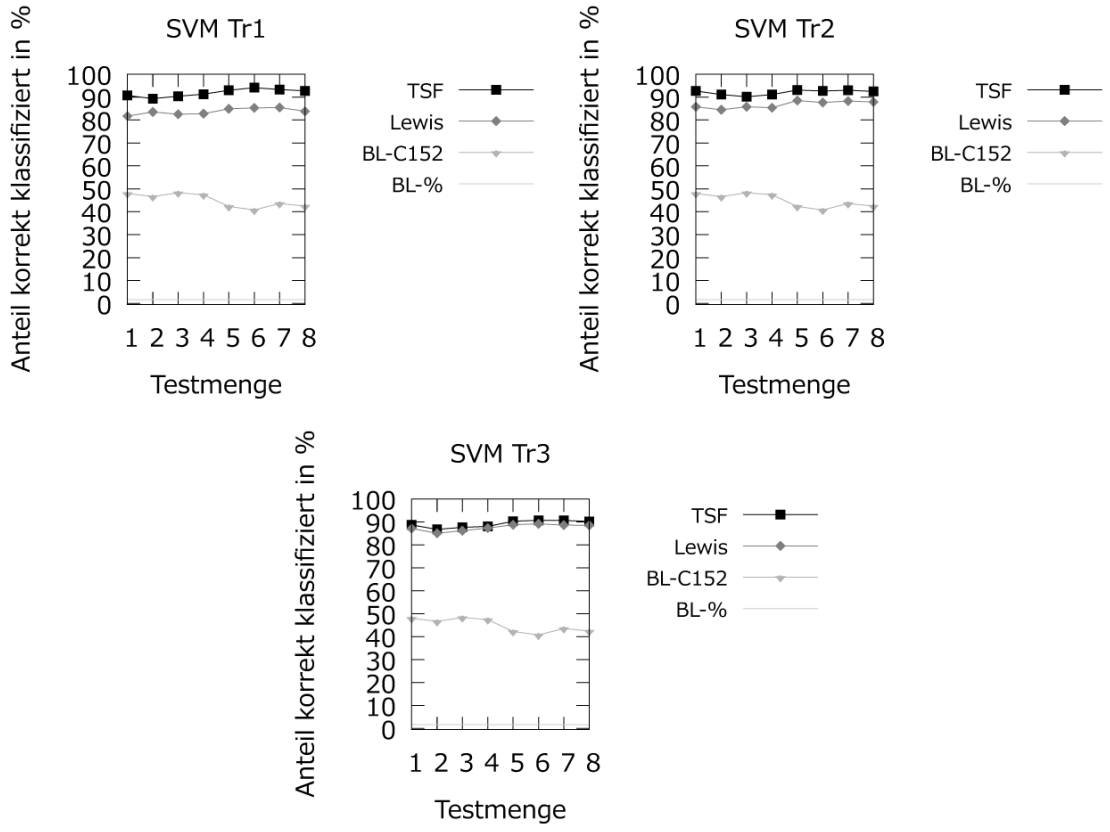


Abbildung 4.77: Ergebnis der Evaluation der Klassifizierung durch den Support-Vector-Machine-Klassifizierer für das Experiment Klass1TV

Auch bei der SVM wird die Annahme getroffen, dass die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen sollte als für Te5 bis Te8. Diese Annahme trifft jedoch für beide Featurevarianten nicht zu. Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von um die 90% richtig klassifizierter Dokumente erreicht wird.

Für beide Verfahren gilt, dass über alle Trainingsteilmengen wesentlich bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung.

– Ergebnisvergleich über alle Algorithmen für das Experiment
Vergleicht man die Ergebnisse des einzelwortbasierten Verfahrens und des Verfahrens mit TSF-Features über alle Algorithmen miteinander, so stellt man Folgendes fest: Das auf TSF-Features basierende Verfahren schneidet beim Klassifizieren von natürlichsprachlichen Dokumenten für fast alle verwendeten Algorithmen besser ab als das einzelwortbasierte Verfahren. Der einzige Algorithmus, für den das nicht zutrifft, ist der Decision Tree. Eine mögliche Ursache für das schlechtere Abschneiden der TSF-Feature-Vektoren wurde bereits im entsprechenden Unterkapitel genannt.

Als Tabelle zusammengefasst ergibt sich für das zweite Experiment folgendes Bild[1]:

Tabelle 4.38: Ergebnisse des Experiments Klass1TV

Verfahren	Algorithmus			
	NB	DT	kNN	SVM
TSF-Feature-Vektoren	✓		✓	✓
Lewis-Feature-Vektoren		✓		

Damit erreichen die TSF-Feature-Vektoren in 5 von 6 Fällen[2] ein besseres Klassifizierungergebnis als die einzelwortbasierten Vektoren.

Somit stützt dieses zweite Experiment die Vermutung, dass wortübergreifende Features qualitativ bessere Ergebnisse beim Klassifizieren von natürlichsprachlichen Dokumenten liefern als Verfahren, die natürlichsprachliche Dokumente in Einzelworte aufteilen und diese als Features für das Klassifizieren benutzen.

4.3.2.1.3 Experiment 3

- ID
  Klass1IV

1 Ein Häkchen zeigt dabei das Verfahren an, das für den betreffenden Algorithmus die höchste Qualität über alle Trainings- und Testteilmengen erreicht hat.
2 Hierbei werden die unterschiedlichen Anzahlen an Nachbarn beim kNN-Algorithmus einzeln gezählt.

- Bezeichnung
  vektorbasiertes Single-label-Industry-Klassifizieren von Testdaten der Reuters-Daten RCV1-v2

- Trainingsdaten
  - 3 randomisiert zusammengestellte Teilmengen der Trainingsdaten des RCV1-v2 mit jeweils 2.000 Dokumenten: Tr1, Tr2, Tr3
  - Dokumente von Tr1 bis Tr3 gehören zu genau einer Industry-Klasse

- Testdaten
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 2.000 Dokumenten: Te1, Te2
  - Dokumente in Te1 und Te2 haben beliebige Klassenzugehörigkeiten, jedoch mindestens eine Industry-Klassenzugehörigkeit[1]
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 10.000 Dokumenten: Te3, Te4
  - Dokumente in Te3 und Te4 haben beliebige Klassenzugehörigkeiten, jedoch mindestens eine Industry-Klassenzugehörigkeit
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 2.000 Dokumenten: Te5, Te6
  - Dokumente in Te5 und Te6 gehören zu genau einer Industry-Klasse
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 10.000 Dokumenten: Te7, Te8
  - Dokumente in Te7 und Te8 gehören zu genau einer Industry-Klasse

- Ähnlichkeit
  vektorbasiert

- Algorithmen
  - Naive Bayes
  - Decision Tree
  - k-Nearest-Neighbour mit $k = 10$, 20 und 200 Nachbarn mit Cosinus zur Ähnlichkeitsberechnung zwischen den Vektoren

1 Zur Erinnerung: Dokumente *können* zu einer oder mehreren Industry-Klassen gehören, *müssen* aber zu keiner Industry-Klasse gehören. Daraus resultiert die Bedingung.

  - Support Vector Machine SVM^{multiclass} mit Parameter $C = 1,0$ und linearem Kernel

- Baseline
  - randomisierte Zuordnung
  - Majority-Class-Zuordnung zu den zwei Klassen mit den meisten Trainingsdokumenten

- Evaluation
  Anteil der richtig klassifizierten Testdaten pro Testteilmenge an der Anzahl der in der Testteilmenge vorhandenen Dokumente

- Ergebnisse[1]
  Für das Klassifizieren der Testmengen Te1 bis Te8 mit dem Naive-Bayes-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.39, 4.40 und 4.41, zu sehen auf S. 359 bis 361.
  Für das Klassifizieren der Testmengen Te1 bis Te8 mit dem Decision-Tree-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.42, 4.43 und 4.44, zu sehen auf S. 361 bis 363.
  Für das Klassifizieren der Testmengen Te1 bis Te8 mit dem k-Nearest-Neighbour-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3 und 10, 20 und 200 Nachbarn, ergeben sich die Ergebnisse aus den Tabellen 4.45, 4.46, 4.47, 4.48, 4.49, 4.50, 4.51, 4.52 und 4.53, zu sehen auf S. 363 bis 369.
  Für das Klassifizieren der Testmengen Te1 bis Te8 mit dem Support-Vector-Machine-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.54, 4.55 und 4.56, zu sehen auf S. 369 bis 371.

Tabelle 4.39: Ergebnisse des Experiments Klass1IV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	325	1675	16,25
	Industry	2	2000	nein	341	1659	17,05
	Industry	3	10000	nein	1682	8318	16,82
wird auf der nächsten Seite fortgesetzt							

1 In allen Tabellen werden die Ergebnisse auf zwei Stellen nach dem Komma gerundet. Alle Klassifizierer sind im abschließenden Vergleich durch ihre Abkürzungen aufgelistet: NB für Naive Bayes, DT für Decision Tree, kNN für k-Nearest-Neighbour und SVM für Support Vector Machine.

Tabelle 4.39: Ergebnisse des Experiments Klass1IV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr1 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Industry	4	10000	nein	1699	8301	16,99
	Industry	5	2000	ja	288	1712	14,40
	Industry	6	2000	ja	313	1687	15,65
	Industry	7	10000	ja	1546	8454	15,46
	Industry	8	10000	ja	1517	8483	15,17
Lewis-Feature-Vektoren	Industry	1	2000	nein	447	1553	22,35
	Industry	2	2000	nein	445	1555	22,25
	Industry	3	10000	nein	2249	7751	22,49
	Industry	4	10000	nein	2256	7744	22,56
	Industry	5	2000	ja	377	1623	18,85
	Industry	6	2000	ja	406	1594	20,30
	Industry	7	10000	ja	2012	7988	20,12
	Industry	8	10000	ja	1969	8031	19,69

Tabelle 4.40: Ergebnisse des Experiments Klass1IV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	307	1693	15,35
	Industry	2	2000	nein	319	1681	15,95
	Industry	3	10000	nein	1578	8422	15,78
	Industry	4	10000	nein	1596	8404	15,96
	Industry	5	2000	ja	257	1743	12,85
	Industry	6	2000	ja	290	1710	14,50
	Industry	7	10000	ja	1443	8557	14,43
	Industry	8	10000	ja	1416	8584	14,16
Lewis-Feature-Vektoren	Industry	1	2000	nein	403	1597	20,15
	Industry	2	2000	nein	427	1573	21,35
	Industry	3	10000	nein	2144	7856	21,44
	Industry	4	10000	nein	2142	7858	21,42
	Industry	5	2000	ja	362	1638	18,10
	Industry	6	2000	ja	369	1631	18,45

wird auf der nächsten Seite fortgesetzt

Tabelle 4.40: Ergebnisse des Experiments Klass1IV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr2 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Industry	7	10000	ja	1909	8091	19,09
	Industry	8	10000	ja	1861	8139	18,61

Tabelle 4.41: Ergebnisse des Experiments Klass1IV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	343	1657	17,15
	Industry	2	2000	nein	356	1644	17,80
	Industry	3	10000	nein	1735	8265	17,35
	Industry	4	10000	nein	1743	8257	17,43
	Industry	5	2000	ja	313	1687	15,65
	Industry	6	2000	ja	340	1660	17,00
	Industry	7	10000	ja	1594	8406	15,94
	Industry	8	10000	ja	1591	8409	15,91
Lewis-Feature-Vektoren	Industry	1	2000	nein	435	1565	21,75
	Industry	2	2000	nein	447	1553	22,35
	Industry	3	10000	nein	2166	7834	21,66
	Industry	4	10000	nein	2145	7855	21,45
	Industry	5	2000	ja	359	1641	17,95
	Industry	6	2000	ja	385	1615	19,25
	Industry	7	10000	ja	1920	8080	19,20
	Industry	8	10000	ja	1906	8094	19,06

Tabelle 4.42: Ergebnisse des Experiments Klass1IV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	252	1748	12,60
	Industry	2	2000	nein	274	1726	13,70
	Industry	3	10000	nein	1358	8642	13,58
wird auf der nächsten Seite fortgesetzt							

Tabelle 4.42: Ergebnisse des Experiments Klass1IV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr1 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Industry	4	10000	nein	1337	8663	13,37
	Industry	5	2000	ja	230	1770	11,50
	Industry	6	2000	ja	245	1755	12,25
	Industry	7	10000	ja	1131	8869	11,31
	Industry	8	10000	ja	1204	8796	12,04
Lewis-Feature-Vektoren	Industry	1	2000	nein	319	1681	15,95
	Industry	2	2000	nein	320	1680	16,00
	Industry	3	10000	nein	1620	8380	16,20
	Industry	4	10000	nein	1614	8386	16,14
	Industry	5	2000	ja	284	1716	14,20
	Industry	6	2000	ja	284	1716	14,20
	Industry	7	10000	ja	1412	8588	14,12
	Industry	8	10000	ja	1398	8602	13,98

Tabelle 4.43: Ergebnisse des Experiments Klass1IV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	298	1702	14,90
	Industry	2	2000	nein	311	1689	15,55
	Industry	3	10000	nein	1482	8518	14,82
	Industry	4	10000	nein	1489	8511	14,89
	Industry	5	2000	ja	251	1749	12,55
	Industry	6	2000	ja	264	1736	13,20
	Industry	7	10000	ja	1341	8659	13,41
	Industry	8	10000	ja	1320	8680	13,20
Lewis-Feature-Vektoren	Industry	1	2000	nein	316	1684	15,80
	Industry	2	2000	nein	338	1662	16,90
	Industry	3	10000	nein	1631	8369	16,31
	Industry	4	10000	nein	1600	8400	16,00
	Industry	5	2000	ja	262	1738	13,10
	Industry	6	2000	ja	281	1719	14,05

wird auf der nächsten Seite fortgesetzt

Tabelle 4.43: Ergebnisse des Experiments Klass1IV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr2 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Industry	7	10000	ja	1453	8547	14,53
	Industry	8	10000	ja	1328	8672	13,28

Tabelle 4.44: Ergebnisse des Experiments Klass1IV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	254	1746	12,70
	Industry	2	2000	nein	264	1736	13,20
	Industry	3	10000	nein	1312	8688	13,12
	Industry	4	10000	nein	1297	8703	12,97
	Industry	5	2000	ja	240	1760	12,00
	Industry	6	2000	ja	255	1745	12,75
	Industry	7	10000	ja	1205	8795	12,05
	Industry	8	10000	ja	1191	8809	11,91
Lewis-Feature-Vektoren	Industry	1	2000	nein	329	1671	16,45
	Industry	2	2000	nein	346	1654	17,30
	Industry	3	10000	nein	1665	8335	16,65
	Industry	4	10000	nein	1663	8337	16,63
	Industry	5	2000	ja	291	1709	14,55
	Industry	6	2000	ja	273	1727	13,65
	Industry	7	10000	ja	1468	8532	14,68
	Industry	8	10000	ja	1452	8548	14,52

Tabelle 4.45: Ergebnisse des Experiments Klass1IV für den 10NN-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	590	1410	29,50
	Industry	2	2000	nein	554	1446	27,70
	Industry	3	10000	nein	2840	7160	28,40
wird auf der nächsten Seite fortgesetzt							

Tabelle 4.45: Ergebnisse des Experiments Klass1IV für den 10NN-Klassifizierer für Trainingsmenge Tr1 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Industry	4	10000	nein	2845	7155	28,45
	Industry	5	2000	ja	468	1532	23,40
	Industry	6	2000	ja	511	1489	25,55
	Industry	7	10000	ja	2472	7528	24,72
	Industry	8	10000	ja	2473	7527	24,73
Lewis-Feature-Vektoren	Industry	1	2000	nein	167	1833	8,35
	Industry	2	2000	nein	168	1832	8,40
	Industry	3	10000	nein	895	9105	8,95
	Industry	4	10000	nein	880	9120	8,80
	Industry	5	2000	ja	168	1832	8,40
	Industry	6	2000	ja	191	1809	9,55
	Industry	7	10000	ja	893	9107	8,93
	Industry	8	10000	ja	845	9155	8,45

Tabelle 4.46: Ergebnisse des Experiments Klass1IV für den 20NN-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	607	1393	30,35
	Industry	2	2000	nein	594	1406	29,70
	Industry	3	10000	nein	2989	7011	29,89
	Industry	4	10000	nein	3033	6967	30,33
	Industry	5	2000	ja	513	1487	25,65
	Industry	6	2000	ja	540	1460	27,00
	Industry	7	10000	ja	2621	7379	26,21
	Industry	8	10000	ja	2615	7385	26,15
Lewis-Feature-Vektoren	Industry	2	2000	nein	168	1832	8,40
	Industry	2	2000	nein	167	1833	8,35
	Industry	3	10000	nein	888	9112	8,88
	Industry	4	10000	nein	886	9114	8,86
	Industry	5	2000	ja	179	1821	8,95
	Industry	6	2000	ja	192	1808	9,60

wird auf der nächsten Seite fortgesetzt

Tabelle 4.46: Ergebnisse des Experiments Klass1IV für den 20NN-Klassifizierer für Trainingsmenge Tr1 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Industry	7	10000	ja	896	9104	8,96
	Industry	8	10000	ja	849	9151	8,49

Tabelle 4.47: Ergebnisse des Experiments Klass1IV für den 200NN-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	611	1389	30,55
	Industry	2	2000	nein	618	1382	30,90
	Industry	3	10000	nein	3066	6934	30,66
	Industry	4	10000	nein	3050	6950	30,50
	Industry	5	2000	ja	502	1498	25,10
	Industry	6	2000	ja	541	1459	27,05
	Industry	7	10000	ja	2661	7339	26,61
	Industry	8	10000	ja	2621	7379	26,21
Lewis-Feature-Vektoren	Industry	1	2000	nein	264	1736	13,20
	Industry	2	2000	nein	270	1730	13,50
	Industry	3	10000	nein	1345	8655	13,45
	Industry	4	10000	nein	1345	8655	13,45
	Industry	5	2000	ja	211	1789	10,55
	Industry	6	2000	ja	245	1755	12,25
	Industry	7	10000	ja	1239	8761	12,39
	Industry	8	10000	ja	1199	8801	11,99

Tabelle 4.48: Ergebnisse des Experiments Klass1IV für den 10NN-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	568	1432	28,40
	Industry	2	2000	nein	570	1430	28,50
	Industry	3	10000	nein	2843	7157	28,43
wird auf der nächsten Seite fortgesetzt							

Tabelle 4.48: Ergebnisse des Experiments Klass1IV für den 10NN-Klassifizierer für Trainingsmenge Tr2 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Industry	4	10000	nein	2813	7187	28,13
	Industry	5	2000	ja	494	1506	24,70
	Industry	6	2000	ja	485	1515	24,25
	Industry	7	10000	ja	2450	7550	24,50
	Industry	8	10000	ja	2440	7560	24,40
Lewis-Feature-Vektoren	Industry	1	2000	nein	188	1812	9,40
	Industry	2	2000	nein	158	1842	7,90
	Industry	3	10000	nein	881	9119	8,81
	Industry	4	10000	nein	893	9107	8,93
	Industry	5	2000	ja	165	1835	8,25
	Industry	6	2000	ja	176	1824	8,80
	Industry	7	10000	ja	827	9173	8,27
	Industry	8	10000	ja	830	9170	8,30

Tabelle 4.49: Ergebnisse des Experiments Klass1IV für den 20NN-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	604	1396	30,20
	Industry	2	2000	nein	574	1426	28,70
	Industry	3	10000	nein	3026	6974	30,26
	Industry	4	10000	nein	2980	7020	29,80
	Industry	5	2000	ja	514	1486	25,70
	Industry	6	2000	ja	511	1489	25,55
	Industry	7	10000	ja	2623	7377	26,23
	Industry	8	10000	ja	2601	7399	26,01
Lewis-Feature-Vektoren	Industry	1	2000	nein	189	1811	9,45
	Industry	2	2000	nein	174	1826	8,70
	Industry	3	10000	nein	926	9074	9,26
	Industry	4	10000	nein	944	9056	9,44
	Industry	5	2000	ja	172	1828	8,60
	Industry	6	2000	ja	184	1816	9,20

wird auf der nächsten Seite fortgesetzt

Tabelle 4.49: Ergebnisse des Experiments Klass1IV für den 20NN-Klassifizierer für Trainingsmenge Tr2 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Industry	7	10000	ja	866	9134	8,66
	Industry	8	10000	ja	864	9136	8,64

Tabelle 4.50: Ergebnisse des Experiments Klass1IV für den 200NN-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	577	1423	28,85
	Industry	2	2000	nein	577	1423	28,85
	Industry	3	10000	nein	2894	7106	28,94
	Industry	4	10000	nein	2882	7118	28,82
	Industry	5	2000	ja	483	1517	24,15
	Industry	6	2000	ja	514	1486	25,70
	Industry	7	10000	ja	2502	7498	25,02
	Industry	8	10000	ja	2492	7508	24,92
Lewis-Feature-Vektoren	Industry	1	2000	nein	255	1745	12,75
	Industry	2	2000	nein	259	1741	12,95
	Industry	3	10000	nein	1294	8706	12,94
	Industry	4	10000	nein	1309	8691	13,09
	Industry	5	2000	ja	207	1793	10,35
	Industry	6	2000	ja	236	1764	11,80
	Industry	7	10000	ja	1202	8798	12,02
	Industry	8	10000	ja	1131	8869	11,31

Tabelle 4.51: Ergebnisse des Experiments Klass1IV für den 10NN-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	581	1419	29,05
	Industry	2	2000	nein	559	1441	27,95
	Industry	3	10000	nein	2860	7140	28,60
wird auf der nächsten Seite fortgesetzt							

Tabelle 4.51: Ergebnisse des Experiments Klass1IV für den 10NN-Klassifizierer für Trainingsmenge Tr3 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Industry	4	10000	nein	2851	7149	28,51
	Industry	5	2000	ja	472	1528	23,60
	Industry	6	2000	ja	511	1489	25,55
	Industry	7	10000	ja	2451	7549	24,51
	Industry	8	10000	ja	2461	7539	24,61
Lewis-Feature-Vektoren	Industry	1	2000	nein	172	1828	8,60
	Industry	2	2000	nein	167	1833	8,35
	Industry	3	10000	nein	895	9105	8,95
	Industry	4	10000	nein	863	9137	8,63
	Industry	5	2000	ja	162	1838	8,10
	Industry	6	2000	ja	180	1820	9,00
	Industry	7	10000	ja	826	9174	8,26
	Industry	8	10000	ja	851	9149	8,51

Tabelle 4.52: Ergebnisse des Experiments Klass1IV für den 20NN-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	610	1390	30,50
	Industry	2	2000	nein	583	1417	29,15
	Industry	3	10000	nein	3013	6987	30,13
	Industry	4	10000	nein	3037	6963	30,37
	Industry	5	2000	ja	496	1504	24,80
	Industry	6	2000	ja	516	1484	25,80
	Industry	7	10000	ja	2604	7396	26,04
	Industry	8	10000	ja	2580	7420	25,80
Lewis-Feature-Vektoren	Industry	1	2000	nein	167	1833	8,35
	Industry	2	2000	nein	187	1813	9,35
	Industry	3	10000	nein	916	9084	9,16
	Industry	4	10000	nein	882	9118	8,82
	Industry	5	2000	ja	164	1836	8,20
	Industry	6	2000	ja	199	1801	9,95

wird auf der nächsten Seite fortgesetzt

Tabelle 4.52: Ergebnisse des Experiments Klass1IV für den 20NN-Klassifizierer für Trainingsmenge Tr3 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Industry	7	10000	ja	898	9102	8,98
	Industry	8	10000	ja	833	9167	8,33

Tabelle 4.53: Ergebnisse des Experiments Klass1IV für den 200NN-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	570	1430	28,50
	Industry	2	2000	nein	557	1443	27,85
	Industry	3	10000	nein	2789	7211	27,89
	Industry	4	10000	nein	2789	7211	27,89
	Industry	5	2000	ja	470	1530	23,50
	Industry	6	2000	ja	492	1508	24,60
	Industry	7	10000	ja	2428	7572	24,28
	Industry	8	10000	ja	2390	7610	23,90
Lewis-Feature-Vektoren	Industry	1	2000	nein	265	1735	13,25
	Industry	2	2000	nein	269	1731	13,45
	Industry	3	10000	nein	1336	8664	13,36
	Industry	4	10000	nein	1336	8664	13,36
	Industry	5	2000	ja	208	1792	10,40
	Industry	6	2000	ja	241	1759	12,05
	Industry	7	10000	ja	1234	8766	12,34
	Industry	8	10000	ja	1178	8822	11,78

Tabelle 4.54: Ergebnisse des Experiments Klass1IV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	750	1250	37,50
	Industry	2	2000	nein	755	1245	37,75
	Industry	3	10000	nein	3755	6245	37,55
wird auf der nächsten Seite fortgesetzt							

Tabelle 4.54: Ergebnisse des Experiments Klass1IV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr1 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Industry	4	10000	nein	3719	6281	37,19
	Industry	5	2000	ja	647	1353	32,35
	Industry	6	2000	ja	662	1338	33,10
	Industry	7	10000	ja	3341	6659	33,41
	Industry	8	10000	ja	3277	6723	32,77
Lewis-Feature-Vektoren	Industry	1	2000	nein	419	1581	20,95
	Industry	2	2000	nein	469	1531	23,45
	Industry	3	10000	nein	2222	7778	22,22
	Industry	4	10000	nein	2228	7772	22,28
	Industry	5	2000	ja	364	1636	18,20
	Industry	6	2000	ja	406	1594	20,30
	Industry	7	10000	ja	2000	8000	20,00
	Industry	8	10000	ja	2059	7941	20,59

Tabelle 4.55: Ergebnisse des Experiments Klass1IV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	737	1263	36,85
	Industry	2	2000	nein	743	1257	37,15
	Industry	3	10000	nein	3743	6257	37,43
	Industry	4	10000	nein	3761	6239	37,61
	Industry	5	2000	ja	666	1334	33,30
	Industry	6	2000	ja	688	1312	34,40
	Industry	7	10000	ja	3351	6649	33,51
	Industry	8	10000	ja	3302	6698	33,02
Lewis-Feature-Vektoren	Industry	1	2000	nein	441	1559	22,05
	Industry	2	2000	nein	442	1558	22,10
	Industry	3	10000	nein	2199	7801	21,99
	Industry	4	10000	nein	2193	7807	21,93
	Industry	5	2000	ja	408	1592	20,40
	Industry	6	2000	ja	387	1613	19,35

wird auf der nächsten Seite fortgesetzt

Tabelle 4.55: Ergebnisse des Experiments Klass1IV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr2 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
	Industry	7	10000	ja	1924	8076	19,24
	Industry	8	10000	ja	1907	8093	19,07

Tabelle 4.56: Ergebnisse des Experiments Klass1IV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	1 Klasse (Test)	Zuordnung		
					richtig	falsch	richtig in %
TSF-Feature-Vektoren	Industry	1	2000	nein	743	1257	37,15
	Industry	2	2000	nein	746	1254	37,30
	Industry	3	10000	nein	3711	6289	37,11
	Industry	4	10000	nein	3678	6322	36,78
	Industry	5	2000	ja	648	1352	32,40
	Industry	6	2000	ja	665	1335	33,25
	Industry	7	10000	ja	3325	6675	33,25
	Industry	8	10000	ja	3262	6738	32,62
Lewis-Feature-Vektoren	Industry	1	2000	nein	389	1611	19,45
	Industry	2	2000	nein	409	1591	20,45
	Industry	3	10000	nein	1989	8011	19,89
	Industry	4	10000	nein	2001	7999	20,01
	Industry	5	2000	ja	321	1679	16,05
	Industry	6	2000	ja	357	1643	17,85
	Industry	7	10000	ja	1736	8264	17,36
	Industry	8	10000	ja	1734	8266	17,34

- Interpretation der Ergebnisse
  - Einführung
    Im Folgenden werden die vorgestellten Ergebnisse für die auf den Suffix Arrays basierenden Vektoren und für die Lewis-Feature-Vektoren einzeln pro Algorithmus miteinander verglichen. Abschließend wird ein allgemeiner Vergleich zwischen den zwei Verfahren bezogen auf das gesamte Experiment über alle Algorithmen und Trainingsteilmengen gezogen.

– Ergebnisvergleich Naive Bayes
Bildet man die Ergebnisse, die mit dem Naive-Bayes-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Folgendes[1]:

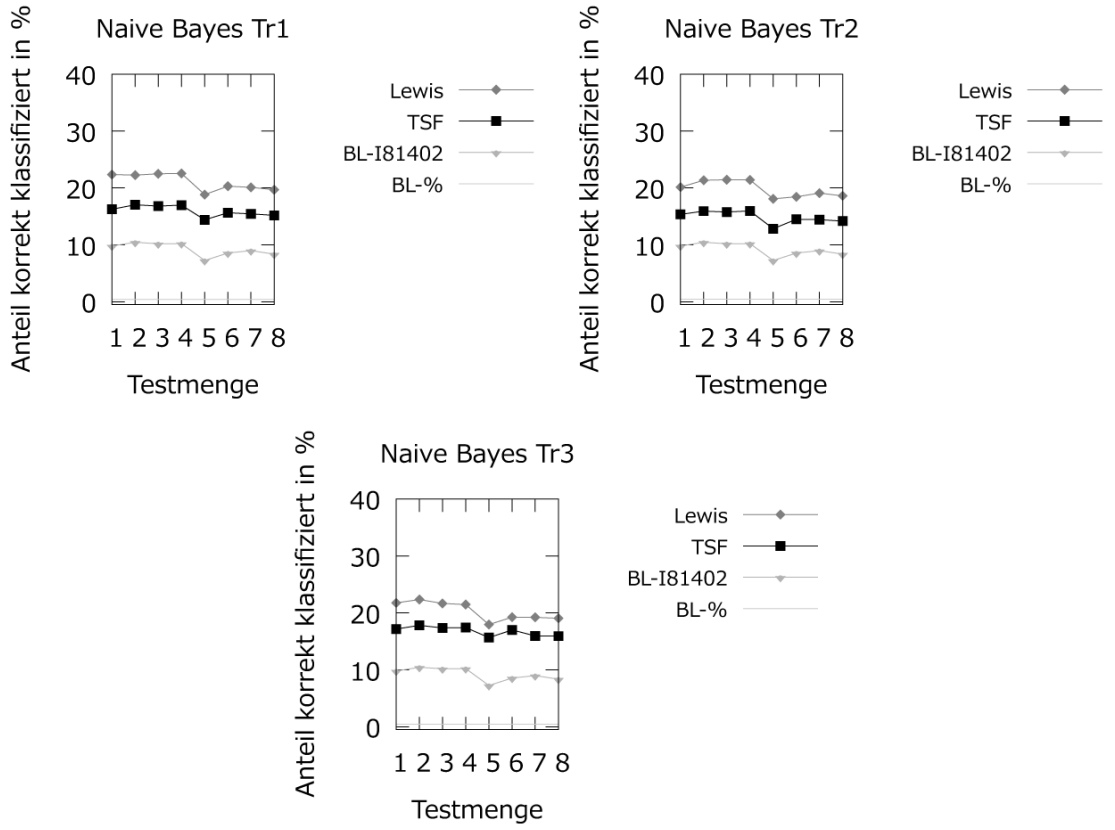


Abbildung 4.78: Ergebnis der Evaluation der Klassifizierung durch den Naive-Bayes-Klassifizierer für das Experiment Klass1IV

Die abgebildeten Ergebnisse zeigen, dass einzelwortbasierte Features für den Naive-Bayes-Klassifizierer und die Klassenfamilie Industry eine bessere Qualität erreichen als die wortübergreifenden Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 4,54 Prozentpunkten zwischen den erreichten Ergebnissen mit den Einzelwort-Features und den TSF-Features. Der minimale Abstand betrug 2,25 Prozentpunkte und der maximale Abstand 6,1 Prozentpunkte.

1 Die Abkürzungen in den Diagrammen bedeuten für dieses Experiment Folgendes: Baseline wird mit „BL“ abgekürzt, „I81402“ steht für den Klassennamen der Klasse mit den meisten Trainingsdokumenten und „%“ steht für die Baseline mit der randomisierten Zuordnung.

Die Trainingsteilmengen bestehen alle drei aus Dokumenten, die nur eine Klassenzuordnung der betrachteten Klassenfamilie *Industry* haben. Um Unterschiede feststellen zu können, wurden in den Testteilmengen zwei verschiedene Auswahlkriterien bezüglich der Klassenzuordnungen der Dokumente getroffen. So ist der erste Teil dieser Teilmengen, die Te1 bis Te4, keiner Einschränkung unterworfen hinsichtlich der Anzahl der Klassenzuordnungen der betrachteten Klassenfamilie, die die enthaltenen Dokumente haben. Das bedeutet, die Dokumente haben mindestens eine Industry-Zuordnung, können aber auch mehrere haben. Der zweite Teil dieser Teilmengen dagegen enthält nur Testdokumente, die *genau eine* Industry-Zuordnung haben. Die Annahme ist, dass, wenn mehr als eine Klasse korrekt sein kann, eine höhere Wahrscheinlichkeit besteht, dass eine der korrekten Klassen zugeordnet wurde. Also sollte die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen als für Te5 bis Te8. Bei beiden Verfahren trifft diese Annahme für die Klassenfamilie Industry für den Naive-Bayes-Algorithmus zu.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von um die 20% richtig klassifizierter Dokumente erreicht wird. Für beide Verfahren gilt, dass über alle Trainingsteilmengen wesentlich bessere Ergebnisse erzielt werden als mit einer zufälligen Zuordnung der Testdokumente zu den zuordenbaren Klassen, d.h., es werden bessere Ergebnisse erreicht als mit der Baseline-Klassifizierung. Ein möglicher Grund für das schlechtere Abschneiden der TSF-Feature-Vektoren gegenüber den einzelwortbasierten Feature-Vektoren beim Naive-Bayes-Klassifizierer für das Experiment Klass1IV könnte sein: Die einzelwortbasierten Feature-Vektoren basieren auf *allen* Trainingsdaten und nicht nur auf einer Teilmenge derselben. Dazu muss man sich noch einmal vor Augen führen, wie die einzelwortbasierten Features ermittelt wurden. Sie stammen aus allen 23.149 Trainingsdokumenten des RCV1-v2-Datensatzes. Das bedeutet, diese Dokumente wurden auf Features untersucht und ihr Vorkommen in den einzelnen Dokumenten durch den entsprechenden Vektor abgebildet. So wird das Vorkommen aller möglichen Features abgebildet, auch derjenigen, die in den von der Verfasserin gebildeten Trainingsteilmengen nicht vorkommen. Die TSF-Feature-Vektoren basieren dagegen nur auf Features, die in den Dokumenten der verwendeten Trainingsteilmengen mindestens doppelt

auftauchen. In diesem Fall wird also eine zweifache Auswahl getroffen: 1. Die Menge der Dokumente, aus denen die TSF-Features stammen, ist kleiner als die Menge der Dokumente, aus denen die Einzelwort-Features stammen, und 2. die TSF-Features müssen mindestens doppelt in dieser Dokumentenmenge auftauchen, um berücksichtigt zu werden. Das wirkt sich im Fall des Naive-Bayes-Algorithmus nachteilig auf die Qualität des Klassifizierungsergebnisses aus.

– Ergebnisvergleich Decision Tree
Bildet man die Ergebnisse, die mit dem Decision-Tree-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Abbildung 4.79 auf S. 375.

Die Ergebnisse des Decision Trees liegen für beide Verfahren nah beieinander. So beträgt der kleinste Abstand zwischen beiden Verfahren 0,08 Prozentpunkte. Als maximaler Abstand für den Decision Tree ergibt sich ein Wert von 4,1 Prozentpunkten. Für die Trainingsteilmengen Tr1 und Tr3 ist das einzelwortbasierte Verfahren besser, bei Tr2 erreichen beide Verfahren fast das gleiche Ergebnis. Im Durchschnitt bedeutet das, dass das einzelwortbasierte Verfahren um 2,15 Prozentpunkte besser abschneidet als das Verfahren mit TSF-Features. Dieser Abstand ist zwar nicht so deutlich, aber er ist vorhanden. Es handelt sich also um den zweiten Algorithmus dieses Experiments für den das einzelwortbasierte Verfahren bessere Ergebnisse erzielt als das Verfahren, welches auf TSF-Features basiert. Eine mögliche Erklärung dafür ist: *overfitting*.[1]

Auch beim Decision Tree wird die Annahme getroffen, dass die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen sollte als für Te5 bis Te8. Diese Annahme trifft beim Decision Tree für beide Featurevarianten zu.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von um die 15% richtig klassifizierter Dokumente erreicht wird. Für beide Verfahren gilt, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung.

1 Siehe S. 58 dieser Arbeit.

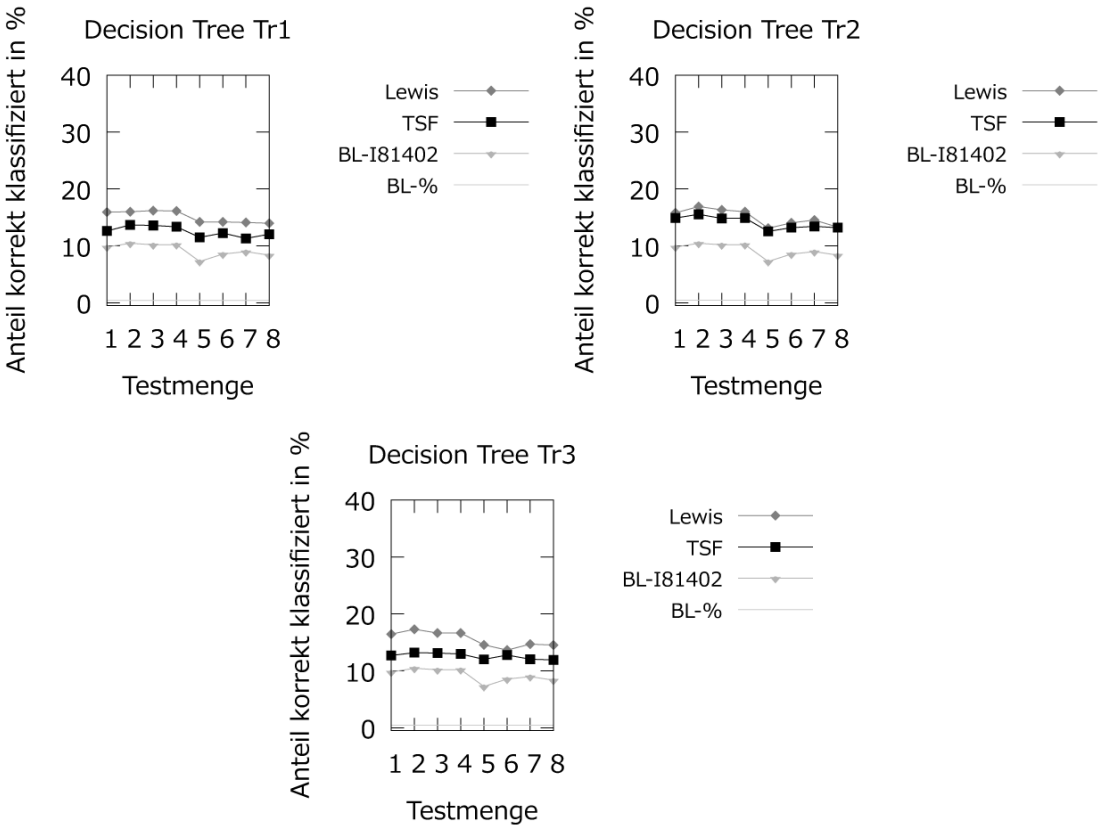


Abbildung 4.79: Ergebnis der Evaluation der Klassifizierung durch den Decision-Tree-Klassifizierer für das Experiment Klass1IV

– Ergebnisvergleich k-Nearest-Neighbour
  Bildet man die Ergebnisse, die mit dem 10-Nearest-Neighbour-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Abbildung 4.80 auf S. 376.

  Die abgebildeten Ergebnisse stützen wieder die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 17,87 Prozentpunkten zwischen den erreichten Ergebnissen mit den TSF-Features und den Einzelwort-Features. Der minimale Abstand betrug 15 Prozentpunkte und der maximale Abstand 21,15 Prozentpunkte. Beim 10NN-Klassifizierer liegen die Ergebnisse beider Verfahren wieder weit auseinander. Auch beim 10NN-Klassifizierer wird die Annahme getroffen, dass die Qualität des

Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen sollte als für Te5 bis Te8. Diese Annahme trifft beim 10NN-Klassifizierer für die TSF-Feature-Vektoren zu. Bei den einzelwortbasierten Feature-Vektoren kann aufgrund der gemischten Ergebnisse keine Aussage darüber getroffen werden.

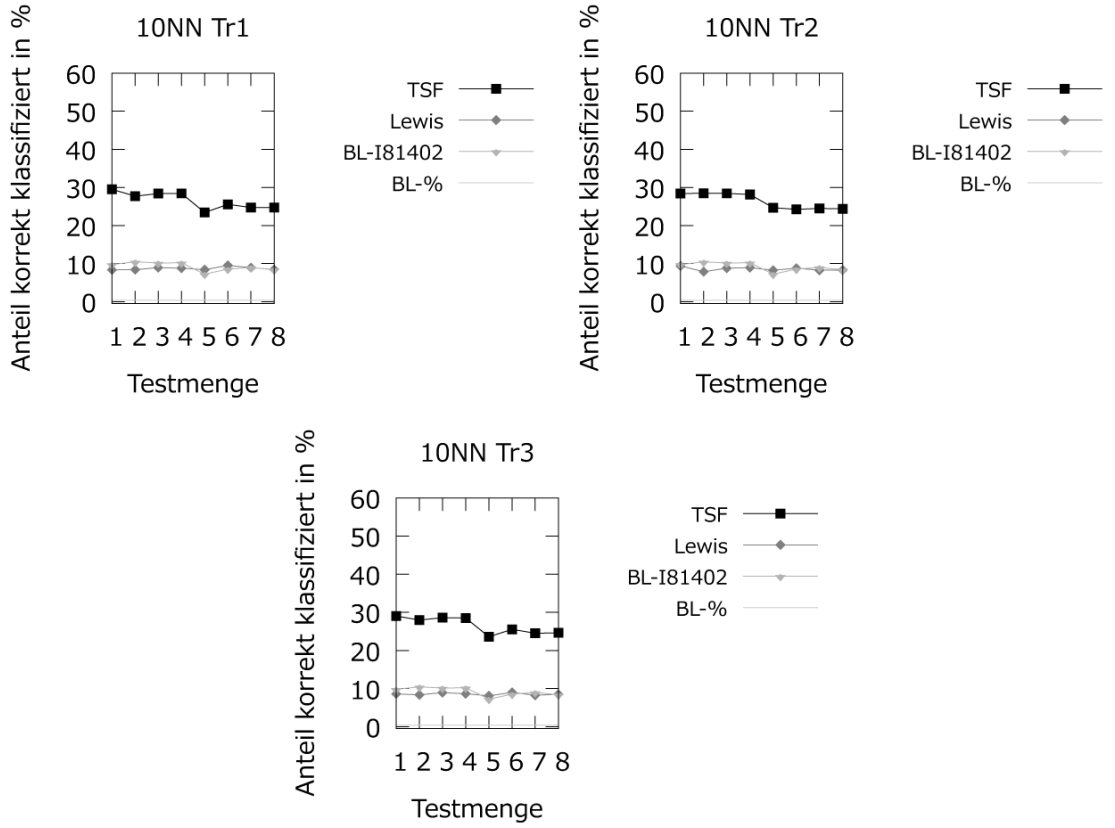


Abbildung 4.80: Ergebnis der Evaluation der Klassifizierung durch den 10-Nearest-Neighbour-Klassifizierer für das Experiment Klass1IV

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von um die 25% richtig klassifizierter Dokumente erreicht wird. Für die TSF-Feature-Vektoren gilt, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung. Die einzelwortbasierten Feature-Vektoren liegen dagegen gleichauf mit der besten Baseline-Klassifizierung.

Bildet man die Ergebnisse, die mit dem 20-Nearest-Neighbour-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Folgendes:

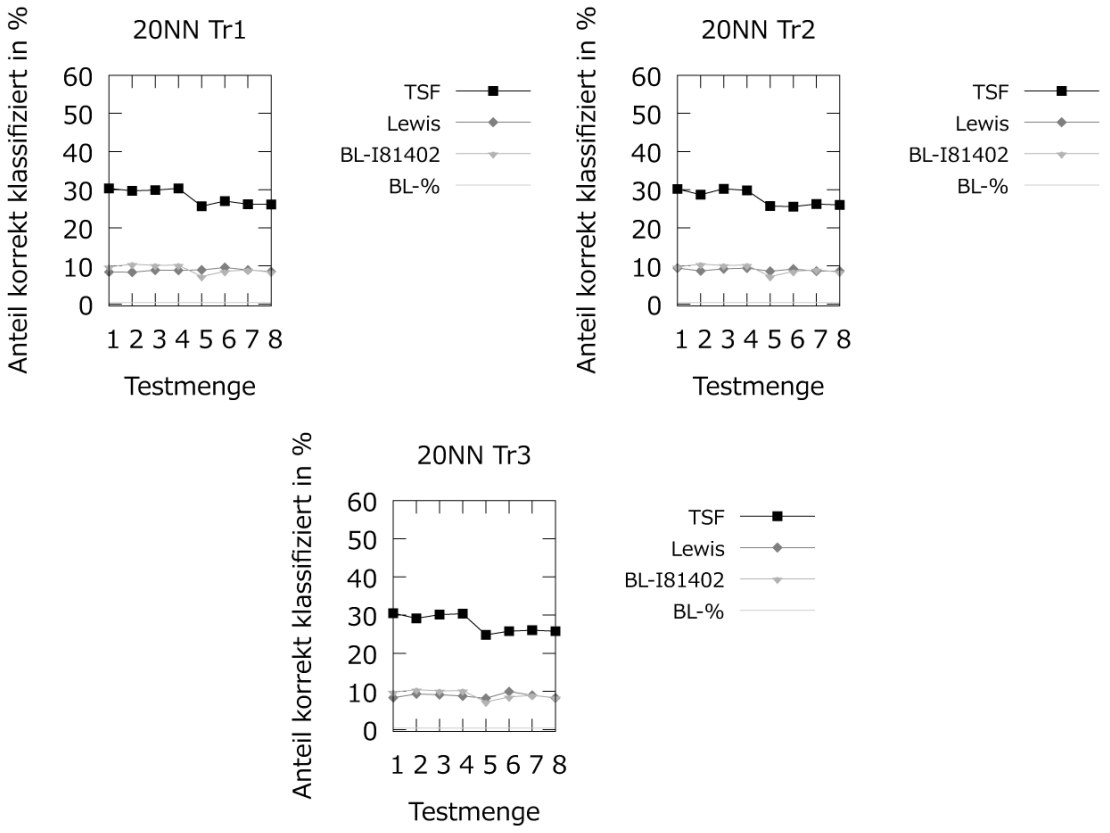


Abbildung 4.81: Ergebnis der Evaluation der Klassifizierung durch den 20-Nearest-Neighbour-Klassifizierer für das Experiment Klass1IV

Die abgebildeten Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 19,03 Prozentpunkten zwischen den erreichten Ergebnissen mit den TSF-Features und den Einzelwort-Features. Das ist der bisher deutlichste gemessene Abstand in diesem Experiment. Der minimale Abstand betrug 15,85 Prozentpunkte und der maximale Abstand 22,15 Prozentpunkte. Beim 20NN-Klassifizierer liegen die Ergebnisse beider Verfahren weit auseinander.

Auch beim 20NN-Klassifizierer wird die Annahme getroffen, dass die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen sollte als für Te5 bis Te8. Diese Annahme trifft beim 20NN-Klassifizierer für die TSF-Feature-Vektoren zu. Bei den einzelwortbasierten Feature-Vektoren kann aufgrund der gemischten Ergebnisse keine Aussage darüber getroffen werden.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von um die 30% richtig klassifizierter Dokumente erreicht wird. Für die TSF-Feature-Vektoren gilt, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung. Die einzelwortbasierten Feature-Vektoren liegen dagegen gleichauf mit der besten Baseline-Klassifizierung.

Bildet man die Ergebnisse, die mit dem 200-Nearest-Neighbour-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Abbildung 4.82 auf S. 379.

Auch diese Ergebnisse stützen die der Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 14,72 Prozentpunkten zwischen den erreichten Ergebnissen mit den TSF-Features und den Einzelwort-Features. Der minimale Abstand betrug 11,94 Prozentpunkte und der maximale Abstand 17,4 Prozentpunkte.

Auch beim 200NN-Klassifizierer wird die Annahme getroffen, dass die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen sollte als für Te5 bis Te8. Diese Annahme trifft für beide Verfahren zu.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal um die 30% richtig klassifizierter Dokumente erreicht wird. Für beide Verfahren gilt, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung.

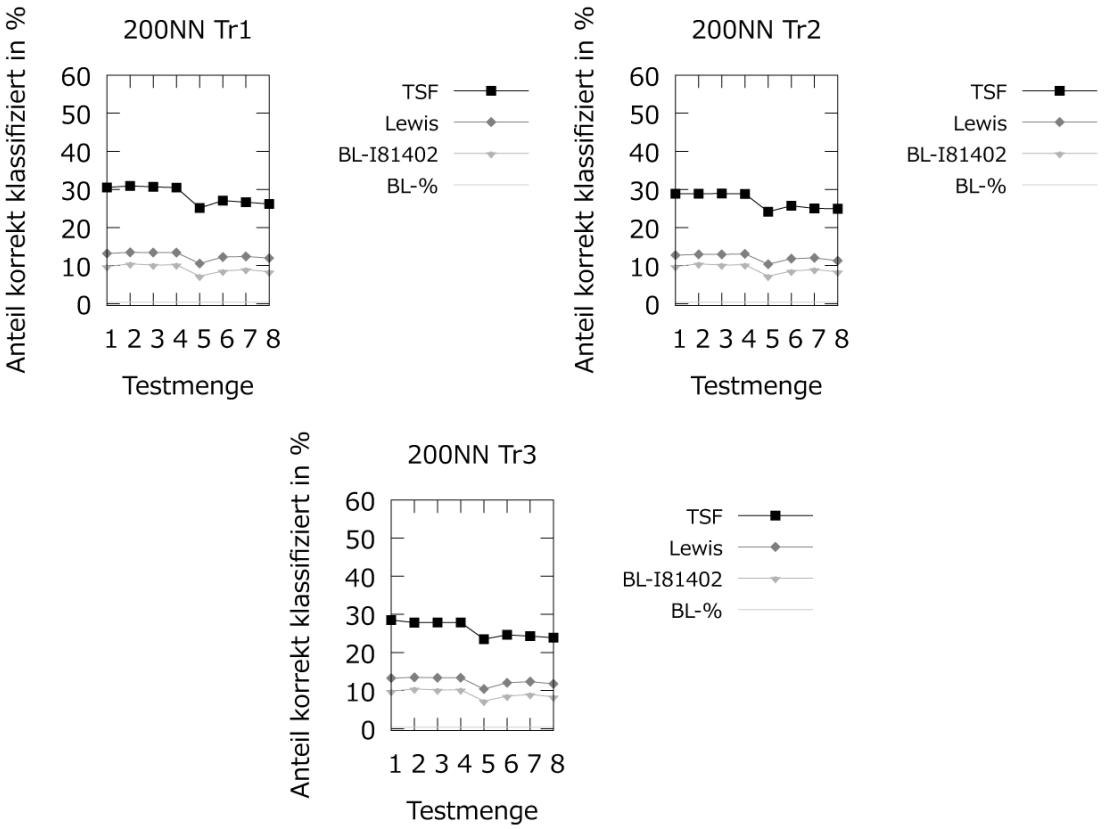


Abbildung 4.82: Ergebnis der Evaluation der Klassifizierung durch den 200-Nearest-Neighbour-Klassifizierer für das Experiment Klass1IV

Insgesamt ergibt sich für den k-Nearest-Neighbour-Klassifizierer folgendes Bild:

* Die TSF-Feature-Vektoren erreichen in allen Fällen bessere Ergebnisse als die einzelwortbasierten Feature-Vektoren.
* Die Anzahl der Nachbarn scheint für die TSF-Feature-Vektoren in diesem Experiment keine Rolle zu spielen, da über alle Anzahlen von Nachbarn ähnliche Qualitätsergebnisse erreicht werden.[1] Bei den einzelwortbasierten Vektoren scheint bei diesem Experiment für den kNN-Algorithmus eine höhere Anzahl von Nachbarn vorteilhafter zu sein.

1 Wahrscheinlich handelt es sich bei der erreichten Qualität um die maximal mögliche für diesen Algorithmus für die Klassenfamilie Industry. Das könnte am Ungleichgewicht zwischen in den Trainingsteilmengen vorhandenen Klassen und in den Testteilmengen vorhandenen Klassen liegen.

– Ergebnisvergleich Support Vector Machine
Bildet man die Ergebnisse, die mit dem Support-Vector-Machine-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Folgendes:

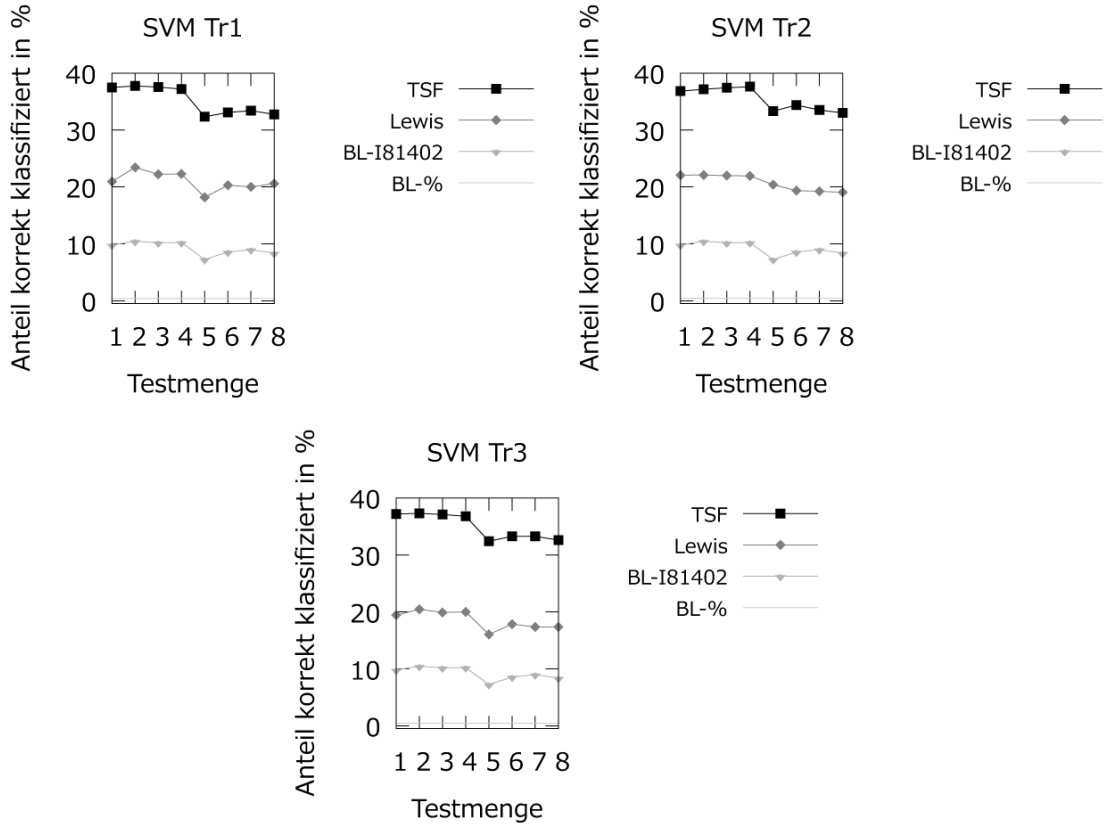


Abbildung 4.83: Ergebnis der Evaluation der Klassifizierung durch den Support-Vector-Machine-Klassifizierer für das Experiment Klass1IV

Die abgebildeten Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 15,09 Prozentpunkten zwischen den erreichten Ergebnissen mit den TSF-Features und den Einzelwort-Features. Der minimale Abstand betrug 12,18 Prozentpunkte und der maximale Abstand 17,7 Prozentpunkte.

Auch bei der SVM wird die Annahme getroffen, dass die Qualität des Ergebnisses für die Testteilmengen Te1 bis Te4 höher liegen sollte als für Te5 bis Te8. Diese Annahme trifft zu.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal um die 35% richtig klassifizierter Dokumente erreicht wird. Für beide Verfahren gilt, dass über alle Trainingsteilmengen wesentlich bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung.

– Ergebnisvergleich über alle Algorithmen für das Experiment
Vergleicht man die Ergebnisse des einzelwortbasierten Verfahrens und des Verfahrens mit TSF-Features über alle Algorithmen miteinander, so stellt man Folgendes fest: Das auf TSF-Features basierende Verfahren schneidet beim Klassifizieren von natürlichsprachlichen Dokumenten für den kNN-Klassifizierer und den SVM-Klassifizierer, die auch absolut besten Klassifizierer, besser ab als das einzelwortbasierte Verfahren. Die beiden Algorithmen, für die das nicht zutrifft, sind der Naive-Bayes-Algorithmus und der Decision-Tree-Algorithmus. Eine mögliche Ursache für das schlechtere Abschneiden der TSF-Feature-Vektoren wurde bereits in den entsprechenden Unterkapiteln genannt.

  Als Tabelle zusammengefasst ergibt sich für das dritte Experiment folgendes Bild[1]:

Tabelle 4.57: Ergebnisse des Experiments Klass1IV

Verfahren	Algorithmus			
	NB	DT	kNN	SVM
TSF-Feature-Vektoren			✓	✓
Lewis-Feature-Vektoren	✓	✓		

Damit erreichen die TSF-Feature-Vektoren in 4 von 6 Fällen ein besseres Klassifizierungergebnis als die einzelwortbasierten Feature-Vektoren.[2]

1 Ein Häkchen zeigt dabei das Verfahren an, das für den betreffenden Algorithmus die höchste Qualität über alle Trainings- und Testteilmengen erreicht hat.

2 Hierbei werden die unterschiedlichen Anzahlen an Nachbarn beim kNN einzeln gezählt.

Somit stützt dieses dritte Experiment die Vermutung, dass wortübergreifende Features qualitativ bessere Ergebnisse beim Klassifizieren von natürlichsprachlichen Dokumenten liefern als Verfahren, die natürlichsprachliche Dokumente in Einzelworte aufteilen und diese als Features für das Klassifizieren benutzen.

4.3.2.1.4 Vergleich der Ergebnisse der Experimente 1 bis 3

Vergleicht man die erreichten Ergebnisse für die TSF-Feature-Vektoren und die wortbasierten Feature-Vektoren über alle Klassenfamilien und Algorithmen miteinander, so erhält man Folgendes:

- In 15 von 18 Fällen erreichen die TSF-Feature-Vektoren bessere Ergebnisse beim Klassifizieren von natürlichsprachlichen Dokumenten als einzelwortbasierte Feature-Vektoren. In Prozent umgerechnet ergibt sich ein Wert von gerundet 83%. Damit kann man, basierend auf diesen Experimenten, sagen, dass die TSF-Feature-Vektoren besser für das Klassifizieren von natürlichsprachlichen Dokumenten geeignet sind als einzelwortbasierte Feature-Vektoren.

- Insbesondere der kNN-Algorithmus, aber auch die SVM scheinen besonders gut mit TSF-Feature-Vektoren klassifizieren zu können, da mit diesen Algorithmen durchweg die besten Ergebnisse erzielt werden. Der Naive-Bayes-Algorithmus erreicht oftmals aufgrund der Zusammensetzung der Trainings- und Testteilmengen schlechtere Ergebnisse für die TSF-Feature-Vektoren und der Decision-Tree-Algorithmus vermutlich aufgrund der Überanpassung an die Trainingsteilmengen.

- Die Annahme, dass ein besseres Klassifizierungsergebnis erreicht wird, wenn mehrere korrekte Klassen in den Trainingsdaten vorhanden sein können, aber trotzdem nur eine zugeordnet wird, wurde zumeist widerlegt. Nur im Fall der Klassenfamilie Industry, konnten die Klassifizierer davon profitieren. Bei den anderen Klassenfamilien war das nicht der Fall.

4.3.2.2 Klassifizierungsexperimente mit dem F-Maß als Evaluationsmaß

4.3.2.2.1 Experiment 4

- ID
  Klass2RV
- Bezeichnung
  vektorbasiertes Single-label-Region-Klassifizieren von Testdaten der Reuters-Daten RCV1-v2
- Trainingsdaten
  - 3 randomisiert zusammengestellte Teilmengen der Trainingsdaten des RCV1-v2 mit jeweils 2.000 Dokumenten: Tr1, Tr2, Tr3[1]
  - Dokumente von Tr1 bis Tr3 gehören zu genau einer Region-Klasse
- Testdaten
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 2.000 Dokumenten: Te5, Te6[2]
  - Dokumente in Te5 und Te6 gehören zu genau einer Region-Klasse
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 10.000 Dokumenten: Te7, Te8
  - Dokumente in Te7 und Te8 gehören zu genau einer Region-Klasse
- Ähnlichkeit
  vektorbasiert
- Algorithmen
  - Naive Bayes
  - Decision Tree
  - k-Nearest-Neighbour mit $k = 10$, 20 und 200 Nachbarn mit Cosinus zur Ähnlichkeitsberechnung zwischen den Vektoren
  - Support Vector Machine SVM^{multiclass} mit Parameter $C = 1,0$ und linearem Kernel

1 Die Trainingsdaten entsprechen den Trainingsdaten des Experiments Klass1RV.
2 Die Testdaten entsprechen den Testdaten des Experiments Klass1RV. Daher beginnt die Nummerierung nicht bei Te1.

- Baseline
  - randomisierte Zuordnung
  - Majority-Class-Zuordnung zu den Klassen mit den meisten Trainingsdokumenten
- Evaluation
  F_1-Maß in der micro- und macroaveraged Form
- Ergebnisse[1]
  Für das Klassifizieren der Testmengen Te5 bis Te8 mit dem Naive-Bayes-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.58, 4.59 und 4.60, zu sehen auf S. 384 bis 385.
  Für das Klassifizieren der Testmengen Te5 bis Te8 mit dem Decision-Tree-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.61, 4.62 und 4.63, zu sehen auf S. 386 bis 386.
  Für das Klassifizieren der Testmengen Te5 bis Te8 mit dem k-Nearest-Neighbour-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3 und 10, 20 und 200 Nachbarn, ergeben sich die Ergebnisse aus den Tabellen 4.64, 4.65 4.66, 4.67, 4.68 4.69, 4.70, 4.71 und 4.72, zu sehen auf S. 387 bis 390.
  Für das Klassifizieren der Testmengen Te5 bis Te8 mit dem Support-Vector-Machine-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.73, 4.74 und 4.75, zu sehen auf S. 390 bis 391.

Tabelle 4.58: Ergebnisse des Experiments Klass2RV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	811	1189	1143	41,02	4,94
	Region	6	2000	826	1174	1134	41,72	5,52
	Region	7	10000	4124	5876	5686	41,64	4,54
	Region	8	10000	4127	5873	5678	41,68	4,64
Lewis-Feature-	Region	5	2000	226	1774	1728	11,43	0,48
	Region	6	2000	200	1800	1760	10,10	0,40

wird auf der nächsten Seite fortgesetzt

1 In allen Tabellen werden die Ergebnisse auf zwei Stellen nach dem Komma gerundet. Alle Klassifizierer sind im abschließenden Vergleich durch ihre Abkürzungen aufgelistet: NB für Naive Bayes, DT für Decision Tree, kNN für k-Nearest-Neighbour und SVM für Support Vector Machine.

Tabelle 4.58: Ergebnisse des Experiments Klass2RV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr1 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
Vektoren	Region	7	10000	1071	8929	8739	10,81	0,44
	Region	8	10000	1093	8907	8712	11,04	0,43

Tabelle 4.59: Ergebnisse des Experiments Klass2RV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	804	1196	1165	40,51	4,07
	Region	6	2000	823	1177	1154	41,39	4,66
	Region	7	10000	4099	5901	5800	41,2	4,26
	Region	8	10000	4111	5889	5746	41,41	3,96
Lewis-Feature-Vektoren	Region	5	2000	245	1755	1724	12,35	0,44
	Region	6	2000	208	1792	1769	10,46	0,39
	Region	7	10000	1146	8854	8753	11,52	0,42
	Region	8	10000	1128	8872	8729	11,36	0,40

Tabelle 4.60: Ergebnisse des Experiments Klass2RV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	787	1213	1174	39,74	3,80
	Region	6	2000	808	1192	1159	40,74	4,43
	Region	7	10000	4054	5946	5787	40,86	4,33
	Region	8	10000	4056	5944	5767	40,92	4,11
Lewis-Feature-Vektoren	Region	5	2000	508	1492	1453	25,65	0,62
	Region	6	2000	522	1478	1445	26,32	0,55
	Region	7	10000	2603	7397	7238	26,24	0,57
	Region	8	10000	2628	7372	7195	26,51	0,56

Tabelle 4.61: Ergebnisse des Experiments Klass2RV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	1184	816	770	59,89	27,14
	Region	6	2000	1236	764	724	62,42	28,99
	Region	7	10000	6028	3972	3782	60,86	27,75
	Region	8	10000	6045	3955	3760	61,05	27,89
Lewis-Feature-Vektoren	Region	5	2000	1140	860	814	57,66	18,88
	Region	6	2000	1119	881	841	56,52	18,83
	Region	7	10000	5619	4381	4191	56,73	18,89
	Region	8	10000	5624	4376	4181	56,79	19,12

Tabelle 4.62: Ergebnisse des Experiments Klass2RV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	1218	782	751	61,38	24,27
	Region	6	2000	1270	730	707	63,88	27,45
	Region	7	10000	6124	3876	3775	61,55	26,79
	Region	8	10000	6151	3849	3706	61,95	26,34
Lewis-Feature-Vektoren	Region	5	2000	1019	981	950	51,35	13,27
	Region	6	2000	1031	969	946	51,85	14,66
	Region	7	10000	5139	4861	4760	51,65	13,91
	Region	8	10000	5211	4789	4646	52,49	14,54

Tabelle 4.63: Ergebnisse des Experiments Klass2RV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	1178	822	783	59,48	26,00
	Region	6	2000	1221	779	746	61,56	26,79
	Region	7	10000	5970	4030	3871	60,18	26,43
	Region	8	10000	6014	3986	3809	60,68	27,02
Lewis-	Region	5	2000	983	1017	978	49,63	12,41
wird auf der nächsten Seite fortgesetzt								

Tabelle 4.63: Ergebnisse des Experiments Klass2RV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr3 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
Feature-Vektoren	Region	6	2000	971	1029	996	48,95	12,46
	Region	7	10000	4878	5122	4963	49,17	11,75
	Region	8	10000	4844	5156	4979	48,87	12,67

Tabelle 4.64: Ergebnisse des Experiments Klass2RV für den 10NN-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	1420	580	534	71,83	35,57
	Region	6	2000	1418	582	542	71,62	34,93
	Region	7	10000	7049	2951	2761	71,17	34,83
	Region	8	10000	7047	2953	2758	71,16	35,57
Lewis-Feature-Vektoren	Region	5	2000	917	1083	1037	46,38	15,93
	Region	6	2000	924	1076	1036	46,67	16,51
	Region	7	10000	4556	5444	5254	46,0	15,20
	Region	8	10000	4629	5371	5176	46,75	16,46

Tabelle 4.65: Ergebnisse des Experiments Klass2RV für den 10NN-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	1455	545	514	73,32	31,14
	Region	6	2000	1449	551	528	72,87	33,47
	Region	7	10000	7150	2850	2749	71,86	32,96
	Region	8	10000	7114	2886	2743	71,65	32,72
Lewis-Feature-Vektoren	Region	5	2000	608	1392	1361	30,64	13,42
	Region	6	2000	616	1384	1361	30,98	13,19
	Region	7	10000	3001	6999	6898	30,16	15,35
	Region	8	10000	3035	6965	6822	30,57	15,28

Tabelle 4.66: Ergebnisse des Experiments Klass2RV für den 10NN-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	1418	582	543	71,6	34,42
	Region	6	2000	1429	571	538	72,04	35,14
	Region	7	10000	7054	2946	2787	71,11	34,77
	Region	8	10000	7021	2979	2802	70,84	34,60
Lewis-Feature-Vektoren	Region	5	2000	924	1076	1037	46,65	16,50
	Region	6	2000	917	1083	1050	46,23	15,83
	Region	7	10000	4589	5411	5252	46,26	18,24
	Region	8	10000	4575	5425	5248	46,16	17,30

Tabelle 4.67: Ergebnisse des Experiments Klass2RV für den 20NN-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	1416	584	538	71,62	33,11
	Region	6	2000	1407	593	553	71,06	32,51
	Region	7	10000	7051	2949	2759	71,19	32,74
	Region	8	10000	7040	2960	2765	71,09	32,23
Lewis-Feature-Vektoren	Region	5	2000	488	1512	1466	24,68	16,26
	Region	6	2000	520	1480	1440	26,26	18,00
	Region	7	10000	2430	7570	7380	24,53	17,57
	Region	8	10000	2527	7473	7278	25,52	18,10

Tabelle 4.68: Ergebnisse des Experiments Klass2RV für den 20NN-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	1444	556	525	72,76	30,17
	Region	6	2000	1444	556	533	72,62	31,96
	Region	7	10000	7141	2859	2758	71,77	30,74
	Region	8	10000	7085	2915	2772	71,36	30,50
Lewis-	Region	5	2000	600	1400	1369	30,23	11,83
wird auf der nächsten Seite fortgesetzt								

Tabelle 4.68: Ergebnisse des Experiments Klass2RV für den 20NN-Klassifizierer für Trainingsmenge Tr2 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
Feature-Vektoren	Region	6	2000	609	1391	1368	30,63	11,95
	Region	7	10000	2923	7077	6976	29,38	13,23
	Region	8	10000	2977	7023	6880	29,98	13,42

Tabelle 4.69: Ergebnisse des Experiments Klass2RV für den 20NN-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	1411	589	550	71,24	33,49
	Region	6	2000	1418	582	549	71,49	32,51
	Region	7	10000	7000	3000	2841	70,56	33,42
	Region	8	10000	6984	3016	2839	70,46	32,60
Lewis-Feature-Vektoren	Region	5	2000	909	1091	1052	45,90	14,77
	Region	6	2000	934	1066	1033	47,09	16,97
	Region	7	10000	4557	5443	5284	45,94	16,48
	Region	8	10000	4546	5454	5277	45,87	16,37

Tabelle 4.70: Ergebnisse des Experiments Klass2RV für den 200NN-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	957	1043	997	48,41	6,33
	Region	6	2000	970	1030	990	48,99	7,41
	Region	7	10000	4830	5170	4980	48,76	6,76
	Region	8	10000	4796	5204	5009	48,43	6,43
Lewis-Feature-Vektoren	Region	5	2000	755	1245	1199	38,19	2,64
	Region	6	2000	756	1244	1204	38,19	2,81
	Region	7	10000	3777	6223	6033	38,13	2,25
	Region	8	10000	3832	6168	5973	38,70	2,74

Tabelle 4.71: Ergebnisse des Experiments Klass2RV für den 200NN-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	942	1058	1027	47,47	5,41
	Region	6	2000	949	1051	1028	47,72	6,04
	Region	7	10000	4719	5281	5180	47,43	5,29
	Region	8	10000	4732	5268	5125	47,66	5,31
Lewis-Feature-Vektoren	Region	5	2000	720	1280	1249	36,28	1,54
	Region	6	2000	745	1255	1232	37,47	1,94
	Region	7	10000	3713	6287	6186	37,32	1,84
	Region	8	10000	3726	6274	6131	37,53	1,76

Tabelle 4.72: Ergebnisse des Experiments Klass2RV für den 200NN-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	951	1049	1010	48,02	6,86
	Region	6	2000	946	1054	1021	47,69	6,94
	Region	7	10000	4736	5264	5105	47,74	6,04
	Region	8	10000	4743	5257	5080	47,85	5,97
Lewis-Feature-Vektoren	Region	5	2000	726	1274	1235	36,66	1,86
	Region	6	2000	739	1261	1228	37,26	2,21
	Region	7	10000	3699	6301	6142	37,29	1,93
	Region	8	10000	3715	6285	6108	37,48	1,89

Tabelle 4.73: Ergebnisse des Experiments Klass2RV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	1297	703	657	65,60	31,59
	Region	6	2000	1313	687	647	66,31	30,25
	Region	7	10000	6515	3485	3295	65,77	30,99
	Region	8	10000	6594	3406	3211	66,59	30,79
Lewis-	Region	5	2000	1117	883	837	56,50	17,93
wird auf der nächsten Seite fortgesetzt								

Tabelle 4.73: Ergebnisse des Experiments Klass2RV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr1 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
Feature-Vektoren	Region	6	2000	1104	896	856	55,76	18,23
	Region	7	10000	5673	4327	4137	57,27	19,31
	Region	8	10000	5614	4386	4191	56,69	18,56

Tabelle 4.74: Ergebnisse des Experiments Klass2RV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	1278	722	691	64,40	25,60
	Region	6	2000	1316	684	661	66,18	27,99
	Region	7	10000	6492	3508	3407	65,25	27,88
	Region	8	10000	6465	3535	3392	65,12	26,72
Lewis-Feature-Vektoren	Region	5	2000	1153	847	816	58,10	15,92
	Region	6	2000	1204	796	773	60,55	18,57
	Region	7	10000	5897	4103	4002	59,27	19,63
	Region	8	10000	5857	4143	4000	58,99	18,65

Tabelle 4.75: Ergebnisse des Experiments Klass2RV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Region	5	2000	1250	750	711	63,12	27,95
	Region	6	2000	1299	701	668	65,49	29,01
	Region	7	10000	6337	3663	3504	63,88	30,43
	Region	8	10000	6372	3628	3451	64,29	29,90
Lewis-Feature-Vektoren	Region	5	2000	1070	930	891	54,03	16,59
	Region	6	2000	1093	907	874	55,10	15,51
	Region	7	10000	5480	4520	4361	55,24	18,67
	Region	8	10000	5451	4549	4372	55,00	17,63

- Interpretation der Ergebnisse
  - Einführung
    Im Folgenden werden die vorgestellten Ergebnisse für die auf den Suffix Arrays basierenden Vektoren und für die Lewis-Feature-Vektoren einzeln pro Algorithmus miteinander verglichen. Abschließend wird ein allgemeiner Vergleich zwischen den zwei Verfahren bezogen auf das gesamte Experiment über alle Algorithmen und Trainingsteilmengen gezogen.
  - Ergebnisvergleich Naive Bayes
    Bildet man die Ergebnisse, die mit dem Naive-Bayes-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Folgendes[1]:

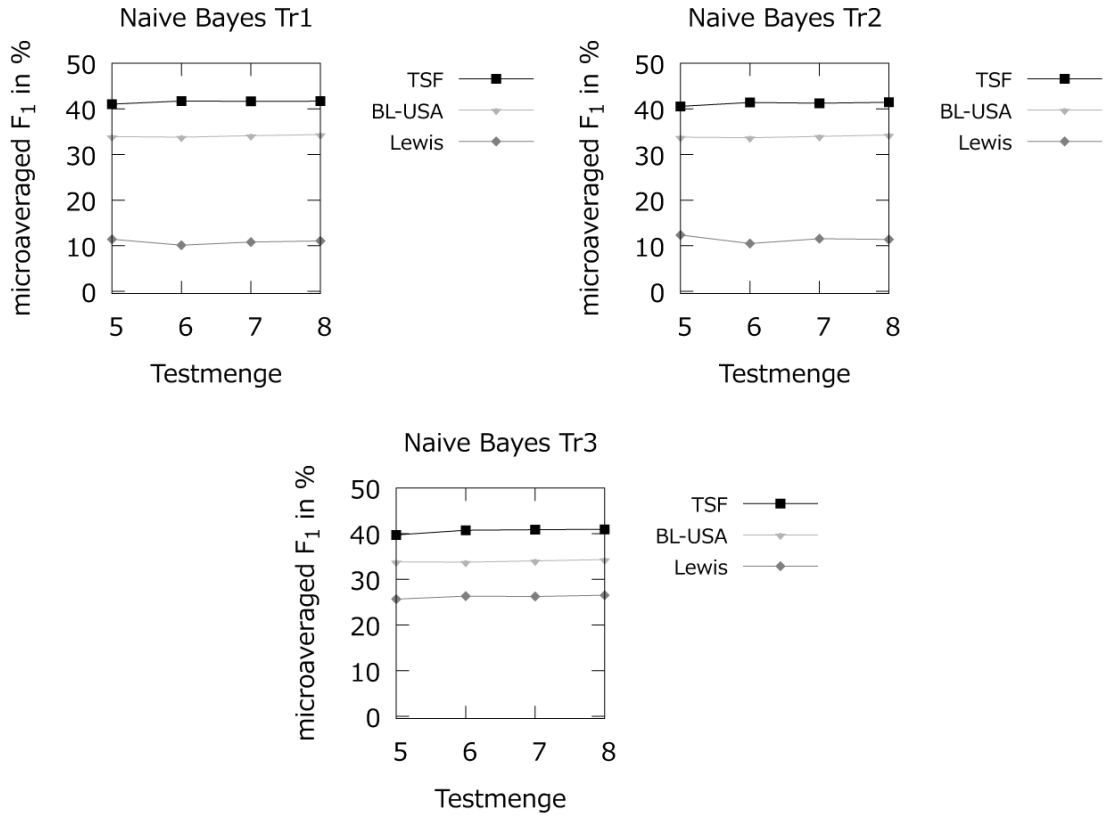


Abbildung 4.84: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den Naive-Bayes-Klassifizierer für das Experiment Klass2RV

1 Die Abkürzungen in den Diagrammen bedeuten für dieses Experiment Folgendes: Baseline wird mit „BL“ abgekürzt und „USA“ steht für den Klassennamen der Klasse mit den meisten Trainingsdokumenten.

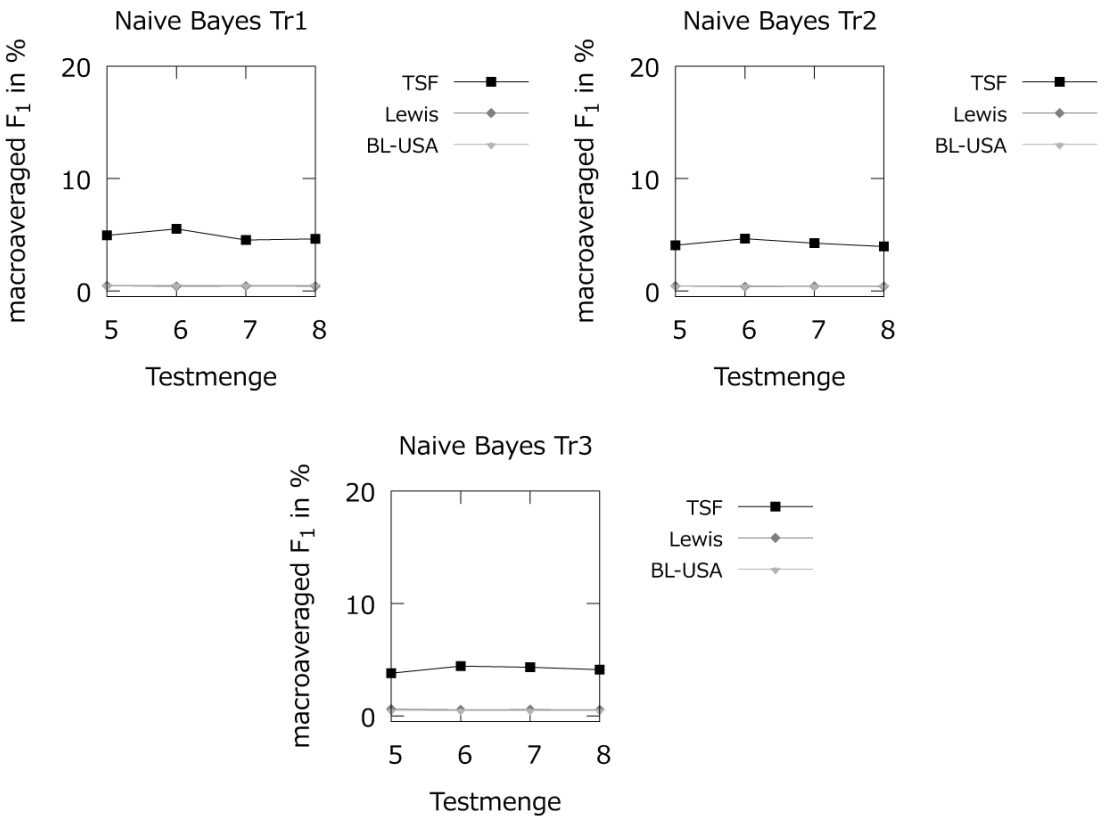


Abbildung 4.85: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den Naive-Bayes-Klassifizierer für das Experiment Klass2RV

Genau wie bei der Evaluation mit dem Anteil der korrekt zugeordneten Dokumente an der Gesamtanzahl an Dokumenten stützen die abgebildeten Ergebnisse die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 24,92 Prozentpunkten zwischen den erreichten Ergebnissen mit den TSF-Features und den Einzelwort-Features bei den microaveraged-Ergebnissen. Der minimale Abstand dort beträgt 14,09 Prozentpunkte und der maximale Abstand 31,62 Prozentpunkte. Legt man also Wert darauf, die Klassifizierungsleistung des Naive Bayes für Klassen mit vielen Dokumenten[1] zu beurteilen, so sind die Ergebnisse für die wortübergreifenden Features besser als die für die einzelwortbasierten Features.

1 Das ist eine microaveraged-Betrachtung, siehe Kapitel 2.5.2.2 auf S. 77 der vorliegenden Arbeit.

Gleiches gilt für die Betrachtung der Leistung bezogen auf die Klassen mit wenigen Dokumenten[1]. Hier klassifiziert der Naive-Bayes-Algorithmus basierend auf wortübergreifenden Features im Schnitt über alle Testteilmengen um 3,39 Prozentpunkte besser als der Naive-Bayes-Algorithmus mit einzelwortbasierten Features. Der minimale Abstand beträgt 3,18 Prozentpunkte und der maximale 5,12 Prozentpunkte. Hier fällt der Unterschied zwischen den beiden Aufbereitungsarten nicht so deutlich aus wie in dem Fall, in dem der Schwerpunkt auf Klassen mit vielen Dokumenten gelegt wird.

Die Ergebnisse der Klassifizierung werden insbesondere bezogen auf den Vergleich zwischen den beiden Verfahren betrachtet. Es folgen hier trotzdem einige Anmerkungen zum absolutem Ergebnis. Es wird eine Qualität von maximal 41% richtig klassifizierten Dokumenten im microaveraged-Fall erreicht. Im macroaveraged-Fall sind es nur 5,5%. Während der Klassifizierer basierend auf den TSF-Feature-Vektoren durchweg, also über beide Auswertungsarten des F_1-Maßes, bessere Ergebnisse erreicht als die Baseline-Klassifizierung, trifft dies auf den Klassifizierer basierend auf Einzelwort-Feature-Vektoren nicht zu. Die mit ihm erreichten Microaveraged-Ergebnisse sind durchweg schlechter als die zufällige Zuordnung der Dokumente zu den Klassen. Im Macroaveraged-Fall entsprechen sich die Ergebnisse der randomisierten Zuordnung und der durch den einzelwortbasierten Klassifizierer erreichten. Somit wäre es im Fall des Naive-Bayes-Klassifizierers basierend auf Einzelwort-Features besser gewesen, den Klassifizierungsvorgang nicht durchzuführen, sondern alle Dokumente direkt der Majority-Klasse zuzuordnen.

– Ergebnisvergleich Decision Tree
Bildet man die Ergebnisse, die mit dem Decision-Tree-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man die Abbildungen 4.86 und 4.87 auf S. 395 und 396.

Die abgebildeten Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen für den Fall, dass der Schwerpunkt bei der Auswertung mit dem F_1-Maß auf Klassen mit vielen Dokumenten gelegt wird, so erhält man im Durchschnitt einen Abstand von 8,6 Prozentpunkten zwischen den Ergeb-

1 Das ist eine macroaveraged-Betrachtung, siehe Kapitel 2.5.2.2 auf S. 77 der vorliegenden Arbeit.

nissen, die mit den TSF-Features und mit den Einzelwort-Features erreicht wurden. Der minimale Abstand beträgt 2,23 Prozentpunkte und der maximale Abstand 12,6 Prozentpunkte. Im Fall der Auswertung des F_1-Maßes mit Schwerpunkt auf Klassen mit wenigen Dokumenten ergibt sich ein ähnliches Bild, jedoch liegen die Ergebnisse hierbei weiter auseinander. So beträgt der minimale Abstand 8,26 Prozentpunkte und der maximale 14,68. Im Durchschnitt sind die Ergebnisse, die mit dem Decision Tree basierend auf den TSF-Feature-Vektoren erreicht werden um 11,79 Prozentpunkte besser als mit dem Decision Tree basierend auf Einzelwort-Feature-Vektoren.

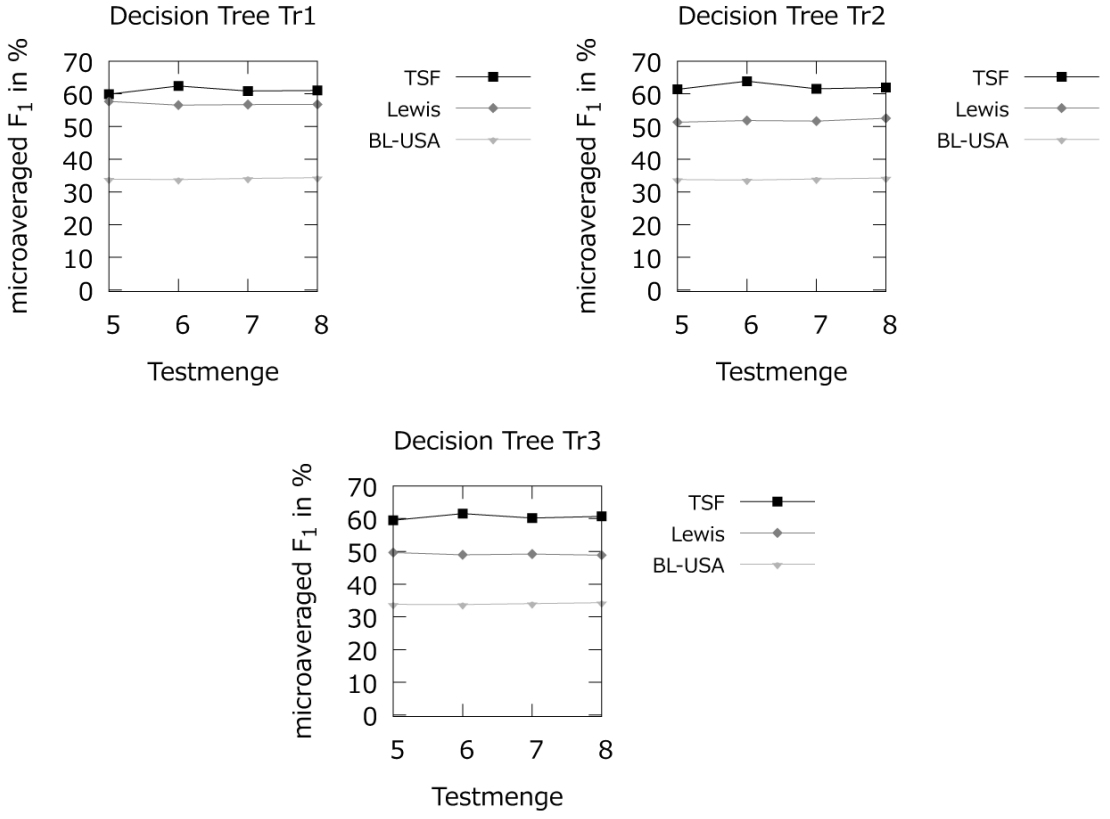


Abbildung 4.86: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den Decision-Tree-Klassifizierer für das Experiment Klass2RV

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal ca. 63% im Microaveraged-Fall erreicht wird und von maximal ca. 28% im Macroaveraged-Fall.

Für beide Verfahren und beide Auswertungsarten des F_1-Maßes gilt, dass über alle Trainingsteilmengen wesentlich bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung.

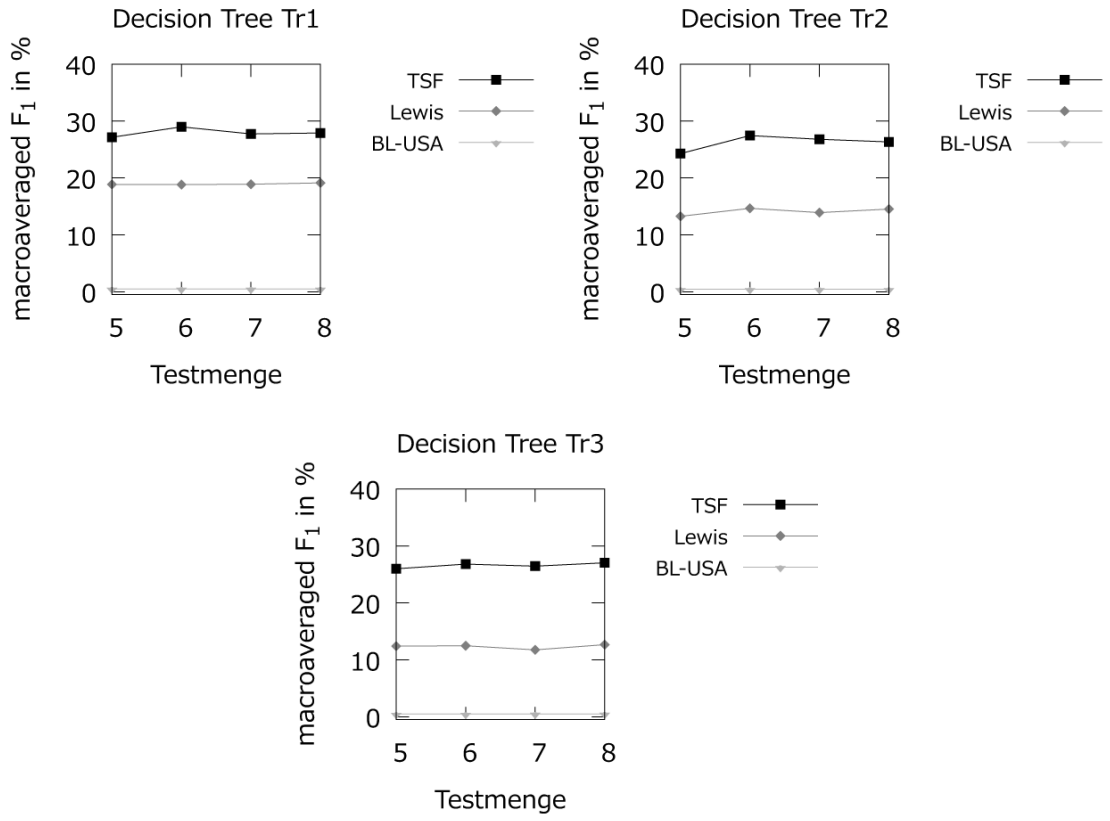


Abbildung 4.87: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den Decision-Tree-Klassifizierer für das Experiment Klass2RV

– Ergebnisvergleich k-Nearest-Neighbour
Bildet man die Ergebnisse, die mit dem 10-Nearest-Neighbour-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man die Abbildungen 4.88 und 4.89 auf S. 397 und 398.

Diese Ergebnisse stützen ebenfalls die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 30,36 Prozentpunkten zwischen den mit den TSF-Features und den Einzelwort-Features erreichten Ergebnissen im Fall der Microaveraged-Auswertung des F_1-Maßes.

Der minimale Abstand beträgt 24,42 Prozentpunkte und der maximale Abstand 42,68 Prozentpunkte.

Wird der Schwerpunkt bei der Auswertung des F_1-Maßes auf Klassen mit wenigen Dokumenten gelegt, so ergibt sich das gleiche Ergebnis.

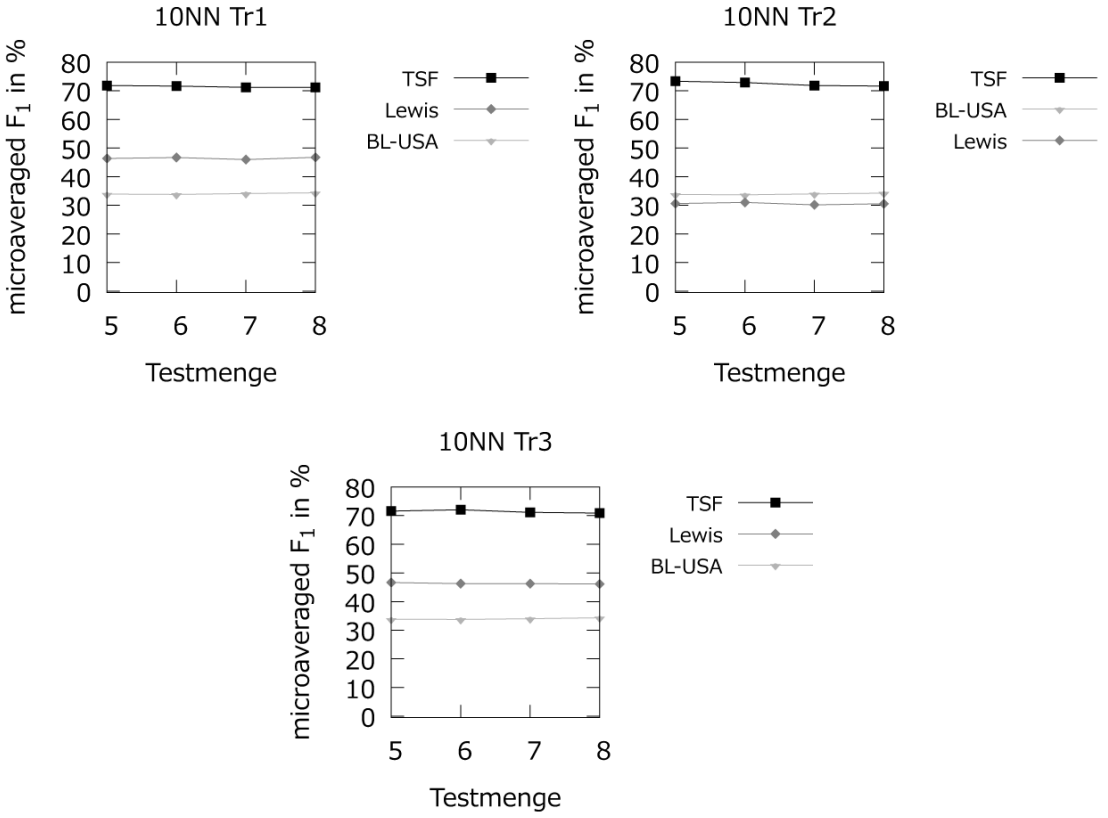


Abbildung 4.88: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den 10-Nearest-Neighbour-Klassifizierer für das Experiment Klass2RV

Der Klassifizierer, der mit TSF-Feature-Vektoren klassifiziert, erreicht durchweg bessere Ergebnisse als der Klassifizierer, der mit Einzelwort-Feature-Vektoren klassifiziert. So unterscheiden sich in diesem Fall die Ergebnisse im Durchschnitt um 18,41 Prozentpunkte, wobei der minimale Abstand 16,52 Prozentpunkte beträgt und der maximale Abstand 20,28 Prozentpunkte.

Im Vordergrund dieser Arbeit steht der Vergleich der Ergebnisse von Klassifizierern, die mit TSF-Feature-Vektoren klassifizieren und Klassifizierern, die mit Einzelwort-Feature-Vektoren klassifizieren. Trotzdem folgen hier einige

Anmerkungen zum absoluten Ergebnis. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal ca. 73% im Microaveraged-Fall erreicht wird. Im Macroaveraged-Fall sind es maximal ca. 35%. Für die TSF-Feature-Vektoren gilt, dass über alle Trainingsteilmengen und beide Auswertungsarten des F_1-Maßes bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung. Für die einzelwortbasierten Feature-Vektoren ist das durchgehend nur bei der Macroaveraged-Auswertung der Fall. Bei der Microaveraged-Auswertung für Tr2 erreicht eine zufällige Zuordnung der Dokumente zur Klasse mit den meisten Trainingsdokumenten eine höhere Qualität.

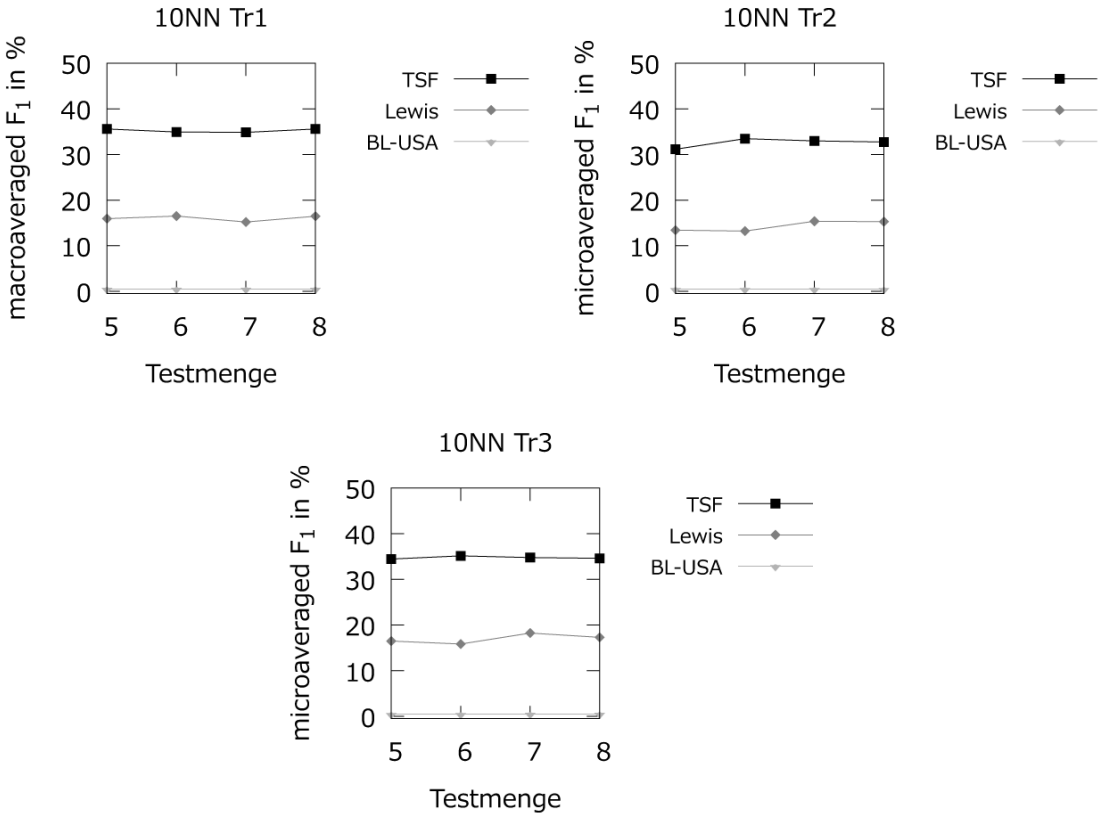


Abbildung 4.89: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den 10-Nearest-Neighbour-Klassifizierer für das Experiment Klass2RV

Bildet man die Ergebnisse, die mit dem 20-Nearest-Neighbour-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man die Abbildungen 4.90 und 4.91 auf S. 399 und 400.

Die abgebildeten Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen für den Microaveraged-Fall, so erhält man im Durchschnitt einen Abstand von 37,6 Prozentpunkten zwischen den mit den TSF-Features und mit den Einzelwort-Features erreichten Ergebnissen. Der minimale Abstand beträgt 24,4 Prozentpunkte und der maximale Abstand 46,94 Prozentpunkte.

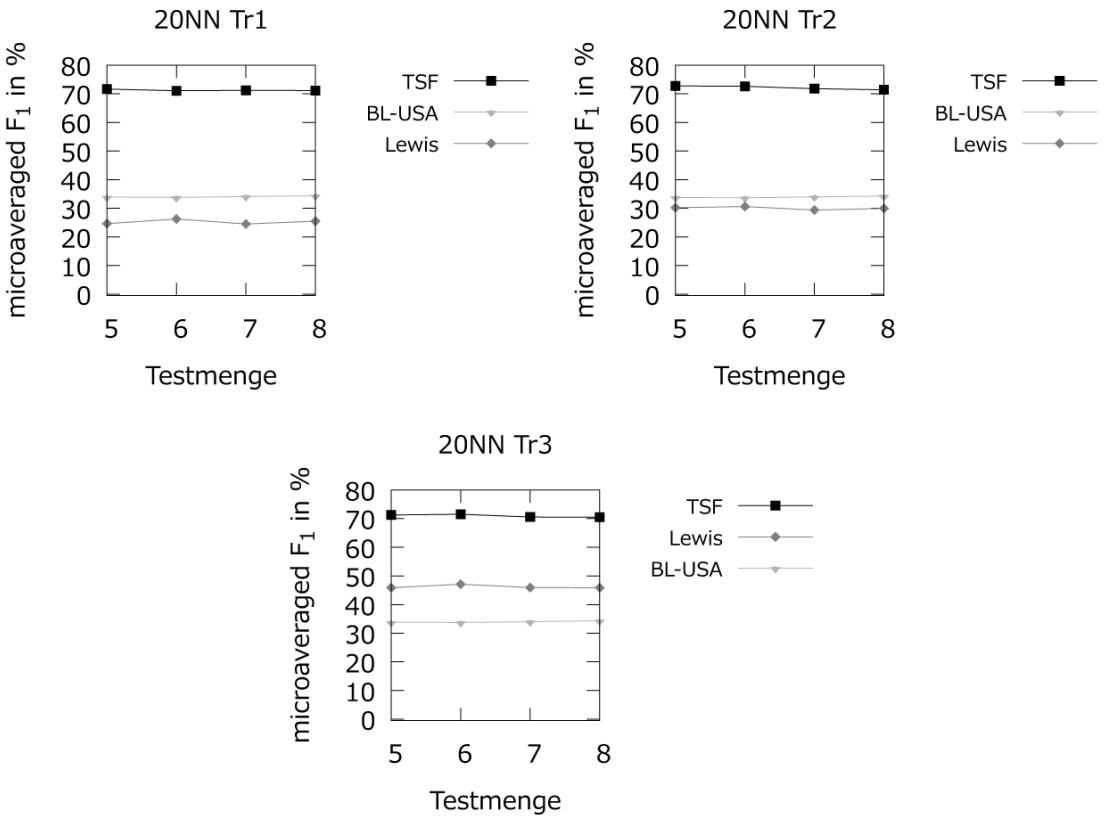


Abbildung 4.90: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den 20-Nearest-Neighbour-Klassifizierer für das Experiment Klass2RV

Auch im Macroaveraged-Fall wird im Durchschnitt ein Abstand von 16,75 Prozentpunkten erreicht, wobei der auf den TSF-Feature-Vektoren basierende Klassifizierer im minimalen Fall um 14,13 Prozentpunkte besser ist als der auf Einzelworten basierende Klassifizierer und im maximalen Fall um 20,01 Prozentpunkte. Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal ca. 72% für den Microaveraged-Fall erreicht wird. Im Macroaveraged-Fall sind es maximal ca. 33%.

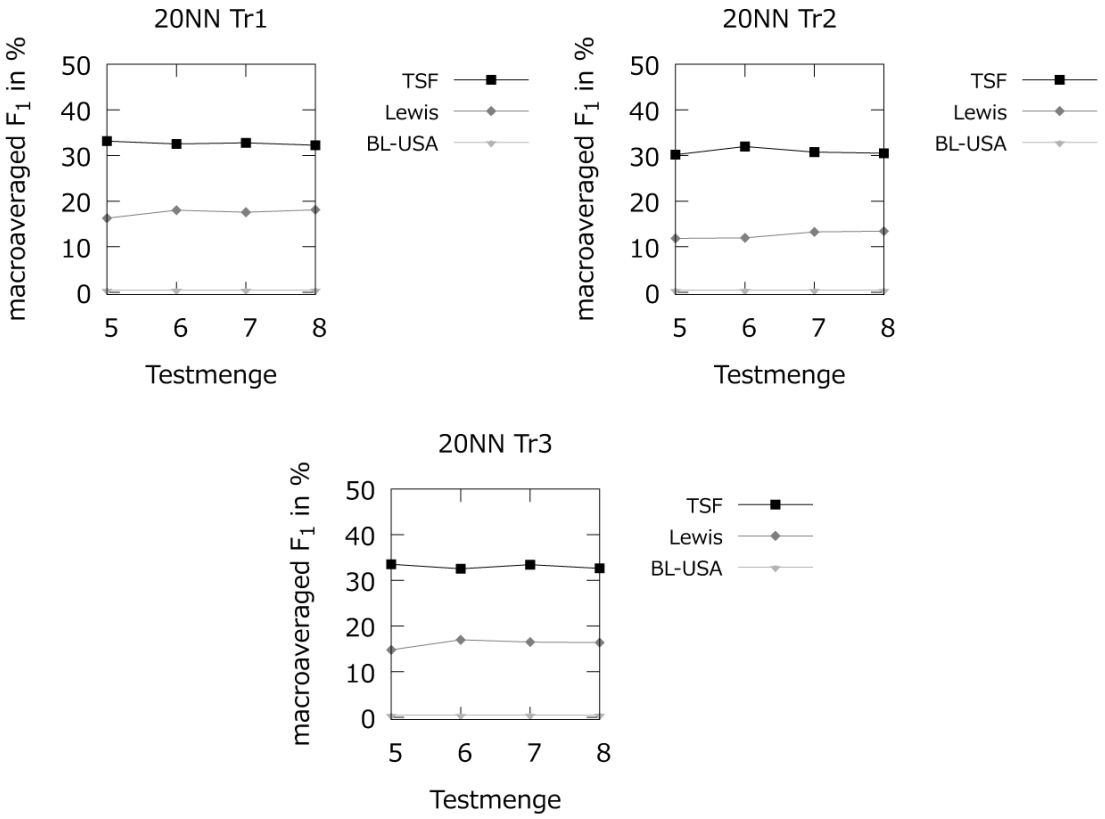


Abbildung 4.91: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den 20-Nearest-Neighbour-Klassifizierer für das Experiment Klass2RV

Für die TSF-Feature-Vektoren gilt, dass über alle Trainingsteilmengen und beide Auswertungsverfahren des F_1-Maßes bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung. Für die einzelwortbasierten Feature-

Vektoren ist das bei Tr1 und Tr2 bei der Microaveraged-Auswertung des F_1-Maßes nicht der Fall. Für diese beiden Trainingsteilmengen wäre eine rein zufällige Zuordnung der Dokumente zur Majority-Klasse besser gewesen.

Bildet man die Ergebnisse, die mit dem 200-Nearest-Neighbour-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man die Abbildungen 4.92 auf dieser Seite und 4.93 auf S. 403.

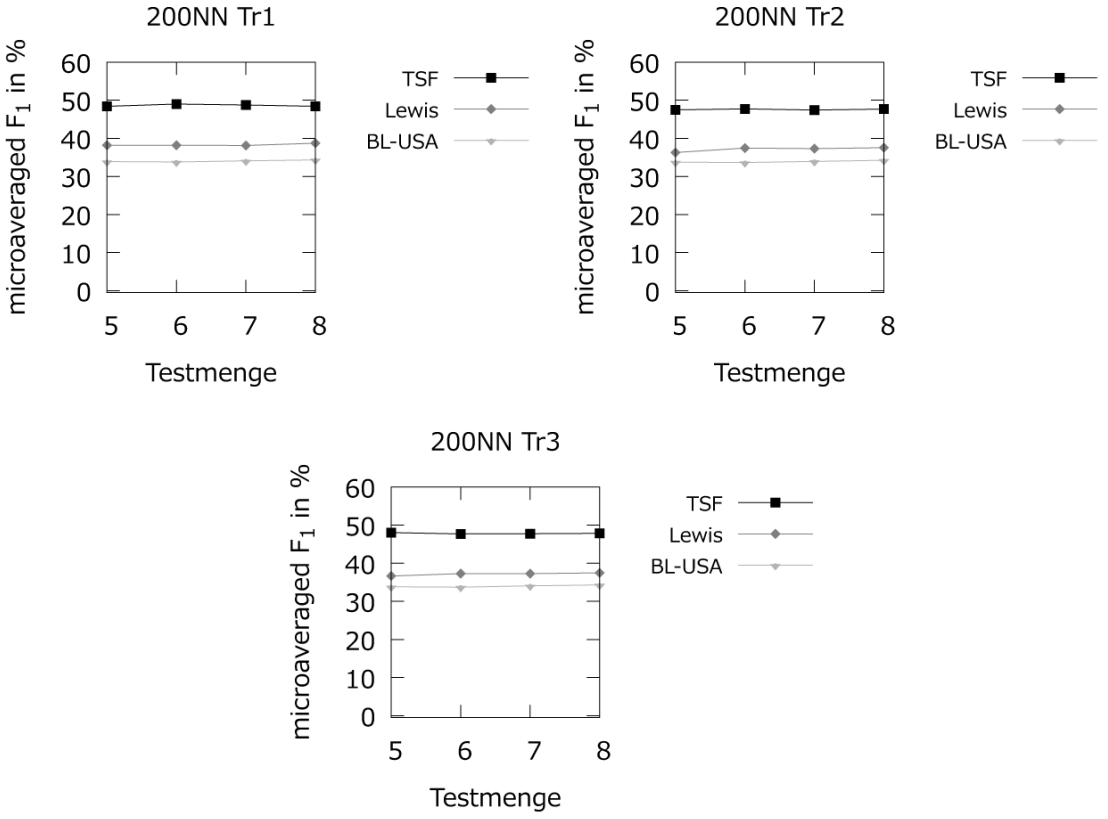


Abbildung 4.92: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den 200-Nearest-Neighbour-Klassifizierer für das Experiment Klass2RV

Auch diese Ergebnisse stützen die der Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt bei der Microaveraged-Auswertung des F_1-Maßes einen Abstand von 10,48 Prozentpunkten zwischen den mit den TSF-Features und mit den Einzelwort-Features erreichten Ergebnissen. Der

minimale Abstand beträgt 9,73 Prozentpunkte und der maximale Abstand 11,36 Prozentpunkte. Der minimale Abstand zwischen den Ergebnissen bei der Macroaveraged-Auswertung des F_1-Maßes beträgt 3,44 Prozentpunkte und der maximale Anstand 5 Prozentpunkte. Im Durchschnitt bedeutet das, dass die Ergebnisse mit TSF-Feature-Vektoren um 4,11 Prozentpunkte über den Ergebnissen mit Einzelwort-Feature-Vektoren liegen.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal ca. 48% im Fall der Microaveraged-Auswertung des F_1-Maßes erreicht wird. Wird der Schwerpunkt der Auswertung des F_1-Maßes auf Klassen mit wenigen Dokumenten gelegt, so sind es maximal ca. 7%.

Für beide Verfahren gilt für beide Auswertungsarten des F_1-Maßes, dass die Ergebnisse der Klassifizierer besser sind als die Baseline-Klassifizierung.

Insgesamt ergibt sich für den k-Nearest-Neighbour-Klassifizierer folgendes Bild:

* Die TSF-Feature-Vektoren erreichen in allen Fällen bessere Ergebnisse als die einzelwortbasierten Feature-Vektoren.
* Die Anzahl der Nachbarn spielt eine entscheidende Rolle für die erreichten Ergebnissen und den Abstand zwischen den TSF-Feature-Vektoren und den einzelwortbasierten Feature-Vektoren.
  Die optimale Anzahl von Nachbarn scheint für das durchgeführte Experiment in der Nähe von 10 zu liegen, da die Ergebnisse mit 20 Nachbarn ein wenig schlechter und mit 200 Nachbarn deutlich schlechter sind, insbesondere das Ergebnis für die Macroaveraged-Auswertung. Das gilt für beide Verfahren.

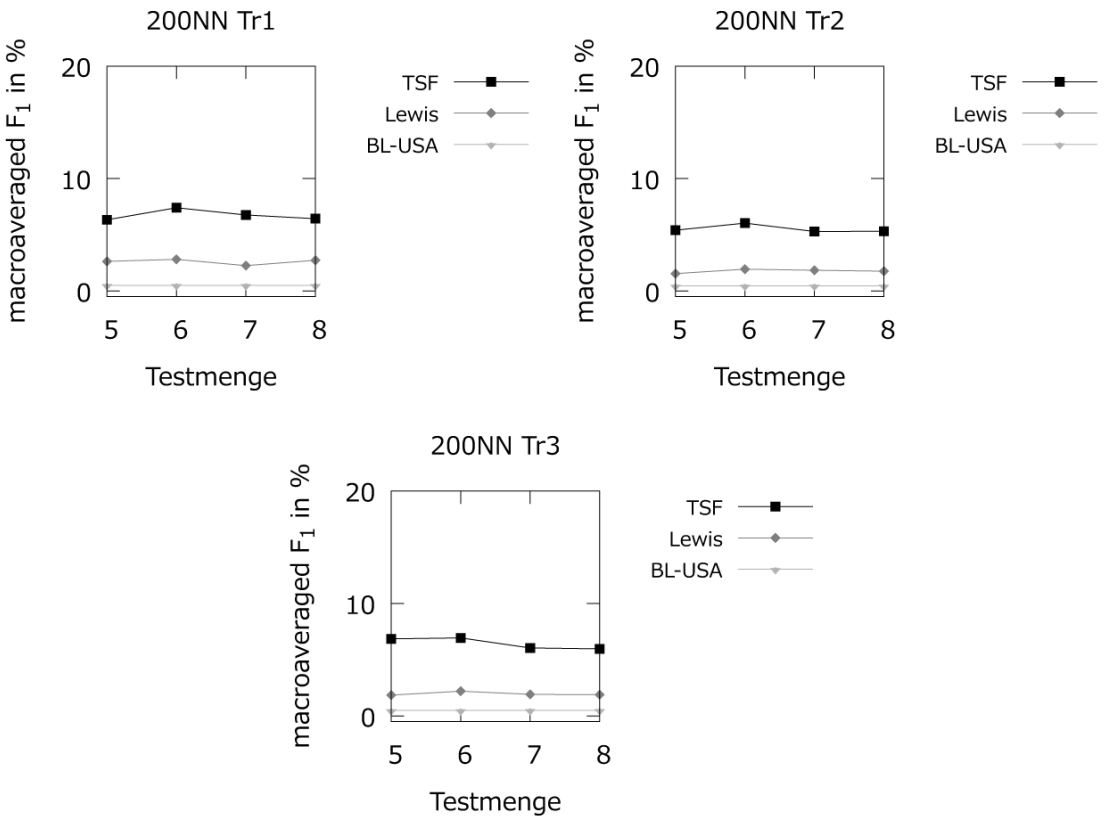


Abbildung 4.93: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den 200-Nearest-Neighbour-Klassifizierer für das Experiment Klass2RV

– Ergebnisvergleich Support Vector Machine
  Bildet man die Ergebnisse, die mit dem Support-Vector-Machine-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man die Abbildungen 4.94 und 4.95 auf S. 405 und 406.

  Die abgebildeten Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Das ist für beide Auswertungsmöglichkeiten des F_1-Maßes der Fall. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen im Fall, dass die Auswertung des F_1-Maßes mit dem Schwerpunkt auf Klassen mit vielen Dokumenten vorgenommen wird, so erhält man im Durchschnitt einen Abstand von 8,29 Prozentpunkten zwischen den mit den TSF-Features und den Einzelwort-Features erreichten Ergebnissen.

Der minimale Abstand beträgt 5,63 Prozentpunkte und der maximale Abstand 10,56 Prozentpunkte. Liegt der Schwerpunkt in der Auswertung des F_1-Maßes auf Klassen mit wenigen Dokumenten, so beträgt der Mindestabstand zwischen den Ergebnissen beider Verfahren 8,07 Prozentpunkte und der Maximalabstand 13,66. Das bedeutet im Durchschnitt erreicht die SVM mit TSF-Feature-Vektoren um 11,16 Prozentpunkte bessere Ergebnisse als die SVM mit Einzelwort-Feature-Vektoren.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal ca. 66% im Microaveraged-Fall erreicht wird. Im Macroaveraged-Fall sind es maximal ca. 31%. Für beide Verfahren gilt, dass über alle Trainingsteilmengen und beide Auswertungsarten des F_1-Maßes wesentlich bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung.

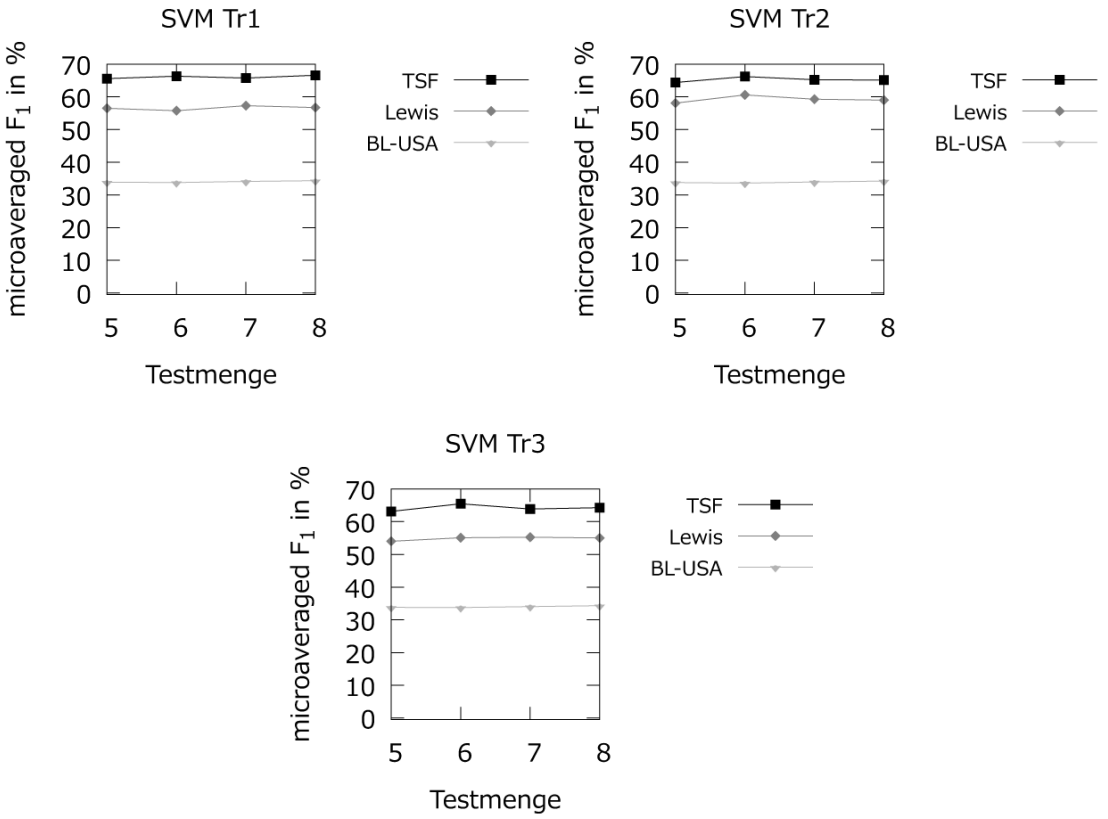


Abbildung 4.94: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den Support-Vector-Machine-Klassifizierer für das Experiment Klass2RV

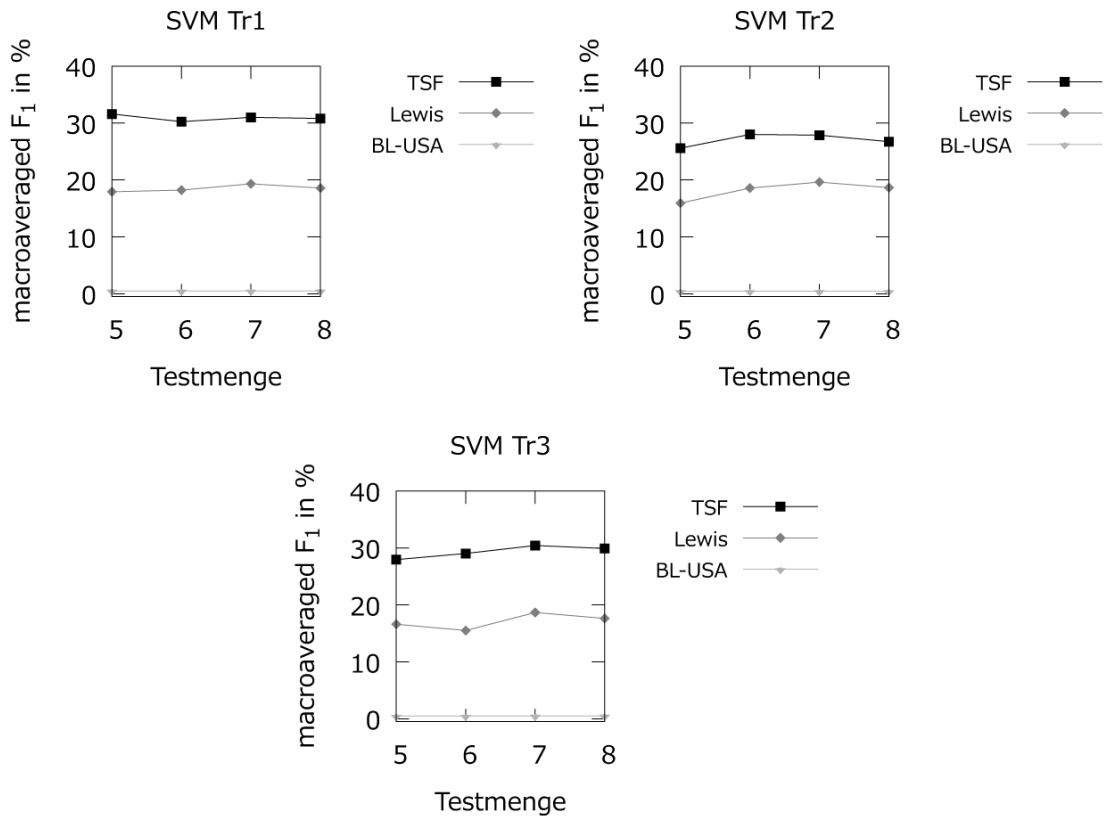


Abbildung 4.95: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den Support-Vector-Machine-Klassifizierer für das Experiment Klass2RV

– Ergebnisvergleich über alle Algorithmen für das Experiment
Vergleicht man die Ergebnisse des einzelwortbasierten Verfahrens und des Verfahrens mit TSF-Features über alle Algorithmen miteinander, so stellt man Folgendes fest: Das auf TSF-Features basierende Verfahren schneidet beim Klassifizieren von natürlichsprachlichen Dokumenten für alle verwendeten Algorithmen besser ab als das einzelwortbasierte Verfahren.

  Somit stützt dieses vierte Experiment die Vermutung, dass wortübergreifende Features qualitativ bessere Ergebnisse beim Klassifizieren von natürlichsprachlichen Dokumenten liefern als Verfahren, die natürlichsprachliche Dokumente in Einzelworte aufteilen und diese als Features für das Klassifizieren benutzen.

Als Tabelle zusammengefasst ergibt sich für das vierte Experiment folgendes Bild[1]:

Tabelle 4.76: Ergebnisse des Experiments Klass2RV

Verfahren	Algorithmus							
	NB (micro)	NB (macro)	DT (micro)	DT (macro)	kNN (micro)	kNN (macro)	SVM (micro)	SVM (macro)
TSF-Feature-Vektoren	✓	✓	✓	✓	✓	✓	✓	✓
Lewis-Feature-Vektoren								

Damit erreichen die TSF-Feature-Vektoren in 12 von 12 Fällen ein besseres Klassifizierungergebnis als die einzelwortbasierten Vektoren.[2]

4.3.2.2.2 Experiment 5

- ID
  Klass2TV
- Bezeichnung
  vektorbasiertes Single-label-Topic-Klassifizieren von Testdaten der Reuters-Daten RCV1-v2
- Trainingsdaten
  - 3 randomisiert zusammengestellte Teilmengen der Trainingsdaten des RCV1-v2 mit jeweils 2.000 Dokumenten: Tr1, Tr2, Tr3[3]
  - Dokumente von Tr1 bis Tr3 gehören zu genau einer Topic-Klasse

1 Ein Häkchen zeigt dabei das Verfahren an, das für den betreffenden Algorithmus die höchste Qualität über alle Trainings- und Testteilmengen erreicht hat.
2 Hierbei werden die unterschiedlichen Anzahlen an Nachbarn beim kNN einzeln gezählt.
3 Die Trainingsdaten entsprechen den Trainingsdaten des Experiments Klass1TV.

- Testdaten
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 2.000 Dokumenten: Te5, Te6[1]
  - Dokumente in Te5 und Te6 gehören zu genau einer Topic-Klasse
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 10.000 Dokumenten: Te7, Te8
  - Dokumente in Te7 und Te8 gehören zu genau einer Topic-Klasse
- Ähnlichkeit
  vektorbasiert
- Algorithmen
  - Naive Bayes
  - Decision Tree
  - k-Nearest-Neighbour mit $k = 10$, 20 und 200 Nachbarn mit Cosinus zur Ähnlichkeitsberechnung zwischen den Vektoren
  - Support Vector Machine SVM^{multiclass} mit Parameter $C = 1,0$ und linearem Kernel
- Baseline
  - randomisierte Zuordnung
  - Majority-Class-Zuordnung zu den zwei Klassen mit den meisten Trainingsdokumenten
- Evaluation
  F_1-Maß in der micro- und macroaveraged Form
- Ergebnisse[2]
  Für das Klassifizieren der Testmengen Te5 bis Te8 mit dem Naive-Bayes-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.77, 4.78 und 4.79, zu sehen auf S. 409 bis 410.

1 Die Testdaten entsprechen den Testdaten des Experiments Klass1TV. Daher beginnt die Numerierung nicht bei Te1.

2 In allen Tabellen werden die Ergebnisse auf zwei Stellen nach dem Komma gerundet. Alle Klassifizierer sind im abschließenden Vergleich durch ihre Abkürzungen aufgelistet: NB für Naive Bayes, DT für Decision Tree, kNN für k-Nearest-Neighbour und SVM für Support Vector Machine.

Für das Klassifizieren der Testmengen Te5 bis Te8 mit dem Decision-Tree-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.80, 4.81 und 4.82, zu sehen auf S. 410 bis 411.
Für das Klassifizieren der Testmengen Te5 bis Te8 mit dem k-Nearest-Neighbour-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3 und 10, 20 und 200 Nachbarn, ergeben sich die Ergebnisse aus den Tabellen 4.83, 4.84 4.85, 4.86, 4.87 4.88, 4.89, 4.90 und 4.91, zu sehen auf S. 411 bis 415.
Für das Klassifizieren der Testmengen Te5 bis Te8 mit dem Support-Vector-Machine-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.92, 4.93 und 4.94, zu sehen auf S. 415 bis 416.

Tabelle 4.77: Ergebnisse des Experiments Klass2TV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	3468	1077	2675	64,90	21,15
	Topic	6	2000	3397	1110	2696	64,09	19,69
	Topic	7	10000	17214	5507	13380	64,57	21,26
	Topic	8	10000	17283	5369	13214	65,04	20,75
Lewis-Feature-Vektoren	Topic	5	2000	3163	1027	3423	58,70	22,97
	Topic	6	2000	3015	1093	3563	56,43	20,65
	Topic	7	10000	15265	5395	17825	56,80	21,35
	Topic	8	10000	15343	5298	17605	57,26	20,89

Tabelle 4.78: Ergebnisse des Experiments Klass2TV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	3406	1184	2822	62,97	19,67
	Topic	6	2000	3402	1159	2747	63,53	19,65
	Topic	7	10000	16987	6025	13982	62,94	19,97
	Topic	8	10000	16996	5892	13866	63,24	19,36
Lewis-Feature-Vektoren	Topic	5	2000	3154	1098	3425	58,24	20,09
	Topic	6	2000	3037	1185	3502	56,44	18,90
	Topic	7	10000	15321	5879	17644	56,57	19,81

wird auf der nächsten Seite fortgesetzt

Tabelle 4.78: Ergebnisse des Experiments Klass2TV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr2 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
	Topic	8	10000	15323	5866	17563	56,67	19,08

Tabelle 4.79: Ergebnisse des Experiments Klass2TV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	3259	1060	3126	60,89	19,33
	Topic	6	2000	3256	1043	3015	61,61	19,05
	Topic	7	10000	16418	5230	15124	61,73	20,19
	Topic	8	10000	16458	5133	15026	62,02	19,45
Lewis-Feature-Vektoren	Topic	5	2000	2962	1005	3826	55,08	19,11
	Topic	6	2000	2840	1065	3962	53,05	18,40
	Topic	7	10000	14400	5361	19636	53,53	19,54
	Topic	8	10000	14552	5153	19319	54,32	19,02

Tabelle 4.80: Ergebnisse des Experiments Klass2TV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	3049	2111	3464	52,24	20,35
	Topic	6	2000	3089	2075	3313	53,42	21,23
	Topic	7	10000	15195	10496	17145	52,37	20,84
	Topic	8	10000	15245	10359	16974	52,73	20,50
Lewis-Feature-Vektoren	Topic	5	2000	3302	1954	2960	57,34	23,99
	Topic	6	2000	3240	1969	3005	56,57	23,56
	Topic	7	10000	16152	9913	15437	56,03	22,22
	Topic	8	10000	16324	9786	14991	56,85	23,24

Tabelle 4.81: Ergebnisse des Experiments Klass2TV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	3083	2088	3433	52,76	19,85
	Topic	6	2000	3022	2162	3435	51,92	19,06
	Topic	7	10000	14957	11100	17800	50,86	19,68
	Topic	8	10000	14995	10911	17499	51,35	19,14
Lewis-Feature-Vektoren	Topic	5	2000	3294	1962	3013	56,97	21,58
	Topic	6	2000	3237	2061	3036	55,95	23,14
	Topic	7	10000	16038	10394	15842	55,01	22,26
	Topic	8	10000	16178	10154	15414	55,86	22,38

Tabelle 4.82: Ergebnisse des Experiments Klass2TV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	3040	2106	3515	51,96	20,98
	Topic	6	2000	3056	2116	3401	52,56	20,62
	Topic	7	10000	15023	10732	17697	51,38	20,43
	Topic	8	10000	14952	10739	17681	51,27	19,79
Lewis-Feature-Vektoren	Topic	5	2000	3313	1878	3008	57,56	23,55
	Topic	6	2000	3203	1966	3152	55,59	23,38
	Topic	7	10000	16309	9701	15404	56,51	23,86
	Topic	8	10000	16404	9528	15139	57,08	23,28

Tabelle 4.83: Ergebnisse des Experiments Klass2TV für den 10NN-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	4039	849	1737	75,75	40,16
	Topic	6	2000	3991	931	1724	75,04	39,53
	Topic	7	10000	19989	4634	8735	74,94	39,95
	Topic	8	10000	20083	4426	8589	75,53	40,25
Lewis-	Topic	5	2000	1847	2658	6030	29,83	17,79
wird auf der nächsten Seite fortgesetzt								

Tabelle 4.83: Ergebnisse des Experiments Klass2TV für den 10NN-Klassifizierer für Trainingsmenge Tr1 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
Feature-Vektoren	Topic	6	2000	1909	2608	5797	31,24	16,53
	Topic	7	10000	9294	13232	29877	30,13	17,21
	Topic	8	10000	9276	13214	29759	30,15	16,54

Tabelle 4.84: Ergebnisse des Experiments Klass2TV für den 10NN-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	4133	792	1587	77,65	39,49
	Topic	6	2000	4002	930	1693	75,32	39,51
	Topic	7	10000	20163	4571	8318	75,78	38,81
	Topic	8	10000	20192	4473	8270	76,01	39,33
Lewis-Feature-Vektoren	Topic	5	2000	2016	2504	5802	32,68	15,93
	Topic	6	2000	2047	2503	5592	33,59	16,12
	Topic	7	10000	10123	12592	28544	32,98	17,05
	Topic	8	10000	9986	12629	28739	32,56	16,73

Tabelle 4.85: Ergebnisse des Experiments Klass2TV für den 10NN-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	3986	891	891	74,43	37,83
	Topic	6	2000	3939	928	1828	74,08	40,38
	Topic	7	10000	19600	4854	9418	73,31	38,38
	Topic	8	10000	19616	4708	9407	73,54	40,17
Lewis-Feature-Vektoren	Topic	5	2000	1973	2503	5877	32,01	16,23
	Topic	6	2000	2000	2486	5698	32,83	17,16
	Topic	7	10000	9988	12546	28832	32,56	17,98
	Topic	8	10000	9991	12496	28746	32,64	17,60

Tabelle 4.86: Ergebnisse des Experiments Klass2TV für den 20NN-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	4047	798	1749	76,06	39,28
	Topic	6	2000	3947	878	1822	74,51	37,59
	Topic	7	10000	19678	4499	9357	73,96	37,52
	Topic	8	10000	19789	4316	9162	74,60	37,23
Lewis-Feature-Vektoren	Topic	5	2000	488	1928	2571	31,17	17,45
	Topic	6	2000	2006	2505	5691	32,86	17,56
	Topic	7	10000	9668	12842	29494	31,35	16,06
	Topic	8	10000	9634	12843	29451	31,30	15,38

Tabelle 4.87: Ergebnisse des Experiments Klass2TV für den 20NN-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	4085	775	1660	77,04	37,03
	Topic	6	2000	4027	857	1653	76,24	36,62
	Topic	7	10000	20055	4438	8574	75,51	36,27
	Topic	8	10000	20050	4389	8584	75,56	36,07
Lewis-Feature-Vektoren	Topic	5	2000	1989	2530	5848	32,19	14,80
	Topic	6	2000	2052	2484	5578	33,73	15,50
	Topic	7	10000	10087	12591	28579	32,89	15,98
	Topic	8	10000	9937	12654	28800	32,41	15,13

Tabelle 4.88: Ergebnisse des Experiments Klass2TV für den 20NN-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	3915	844	1989	73,43	34,35
	Topic	6	2000	3861	886	1966	73,03	34,85
	Topic	7	10000	19313	4613	9970	72,59	35,31
	Topic	8	10000	19325	4507	9947	72,78	34,96
Lewis-	Topic	5	2000	1965	2502	5897	31,88	15,72
wird auf der nächsten Seite fortgesetzt								

Tabelle 4.88: Ergebnisse des Experiments Klass2TV für den 20NN-Klassifizierer für Trainingsmenge Tr3 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
Feature-Vektoren	Topic	6	2000	1990	2499	5709	32,66	16,39
	Topic	7	10000	9868	12643	29042	32,13	15,66
	Topic	8	10000	9876	12574	28939	32,24	15,81

Tabelle 4.89: Ergebnisse des Experiments Klass2TV für den 200NN-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	3414	1121	2784	63,62	18,14
	Topic	6	2000	3280	1152	2914	61,74	17,59
	Topic	7	10000	16447	5955	14753	61,37	17,14
	Topic	8	10000	16561	5768	14554	61,98	17,51
Lewis-Feature-Vektoren	Topic	5	2000	2134	3759	4838	33,18	6,92
	Topic	6	2000	2046	3833	4963	31,75	6,52
	Topic	7	10000	10605	18784	24137	33,07	6,57
	Topic	8	10000	10349	19008	24543	32,22	6,42

Tabelle 4.90: Ergebnisse des Experiments Klass2TV für den 200NN-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	3479	1134	2678	64,61	19,40
	Topic	6	2000	3321	1210	2835	62,15	18,13
	Topic	7	10000	16758	6209	14153	62,21	17,55
	Topic	8	10000	16835	5997	14002	62,74	17,82
Lewis-Feature-Vektoren	Topic	5	2000	2194	3686	4737	34,25	7,59
	Topic	6	2000	2093	3769	4866	32,65	7,69
	Topic	7	10000	10941	18372	23535	34,30	7,65
	Topic	8	10000	10602	18729	24092	33,12	7,41

Tabelle 4.91: Ergebnisse des Experiments Klass2TV für den 200NN-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	3228	1083	3203	60,10	17,69
	Topic	6	2000	3164	1092	3201	59,58	17,28
	Topic	7	10000	15811	5665	16241	59,08	16,83
	Topic	8	10000	15851	5474	16189	59,41	16,96
Lewis-Feature-Vektoren	Topic	5	2000	2184	3658	4760	34,16	8,06
	Topic	6	2000	2053	3783	4959	31,96	7,33
	Topic	7	10000	10808	18425	23761	33,88	7,60
	Topic	8	10000	10702	18499	23912	33,54	7,91

Tabelle 4.92: Ergebnisse des Experiments Klass2TV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	4158	904	1537	77,31	37,49
	Topic	6	2000	4037	1031	1645	75,11	35,29
	Topic	7	10000	20382	5037	8171	75,53	34,88
	Topic	8	10000	20466	4854	7949	76,17	35,49
Lewis-Feature-Vektoren	Topic	5	2000	3648	1313	2470	65,85	29,51
	Topic	6	2000	3546	1404	2579	64,04	29,70
	Topic	7	10000	17997	6944	12593	64,82	27,94
	Topic	8	10000	17803	7003	12982	64,05	27,48

Tabelle 4.93: Ergebnisse des Experiments Klass2TV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	4135	924	1585	76,72	34,66
	Topic	6	2000	4000	1054	1730	74,18	32,38
	Topic	7	10000	20343	5098	8188	75,38	34,17
	Topic	8	10000	20369	4967	8104	75,71	33,81
Lewis-	Topic	5	2000	3512	1776	2603	61,60	26,31
wird auf der nächsten Seite fortgesetzt								

Tabelle 4.93: Ergebnisse des Experiments Klass2TV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr2 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
Feature-Vektoren	Topic	6	2000	3436	1837	2651	60,49	26,99
	Topic	7	10000	17421	9049	13017	61,23	26,31
	Topic	8	10000	17517	8811	12832	61,81	26,22

Tabelle 4.94: Ergebnisse des Experiments Klass2TV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Topic	5	2000	4077	952	1709	75,40	33,55
	Topic	6	2000	3971	1063	1807	73,46	33,49
	Topic	7	10000	20014	5181	8928	73,94	34,05
	Topic	8	10000	20068	5035	8776	74,40	32,68
Lewis-Feature-Vektoren	Topic	5	2000	3796	1265	2149	68,98	30,30
	Topic	6	2000	3741	1325	2167	68,18	29,45
	Topic	7	10000	18672	6690	11042	67,80	29,60
	Topic	8	10000	18707	6579	10963	68,08	29,45

- Interpretation der Ergebnisse
  - Einführung
    Im Folgenden werden die vorgestellten Ergebnisse für die auf den Suffix Arrays basierenden Vektoren und für die Lewis-Feature-Vektoren einzeln pro Algorithmus miteinander verglichen. Abschließend wird ein allgemeiner Vergleich zwischen den zwei Verfahren bezogen auf das gesamte Experiment über alle Algorithmen und Trainingsteilmengen gezogen.

– Ergebnisvergleich Naive Bayes
  Bildet man die Ergebnisse, die mit dem Naive-Bayes-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Folgendes[1]:

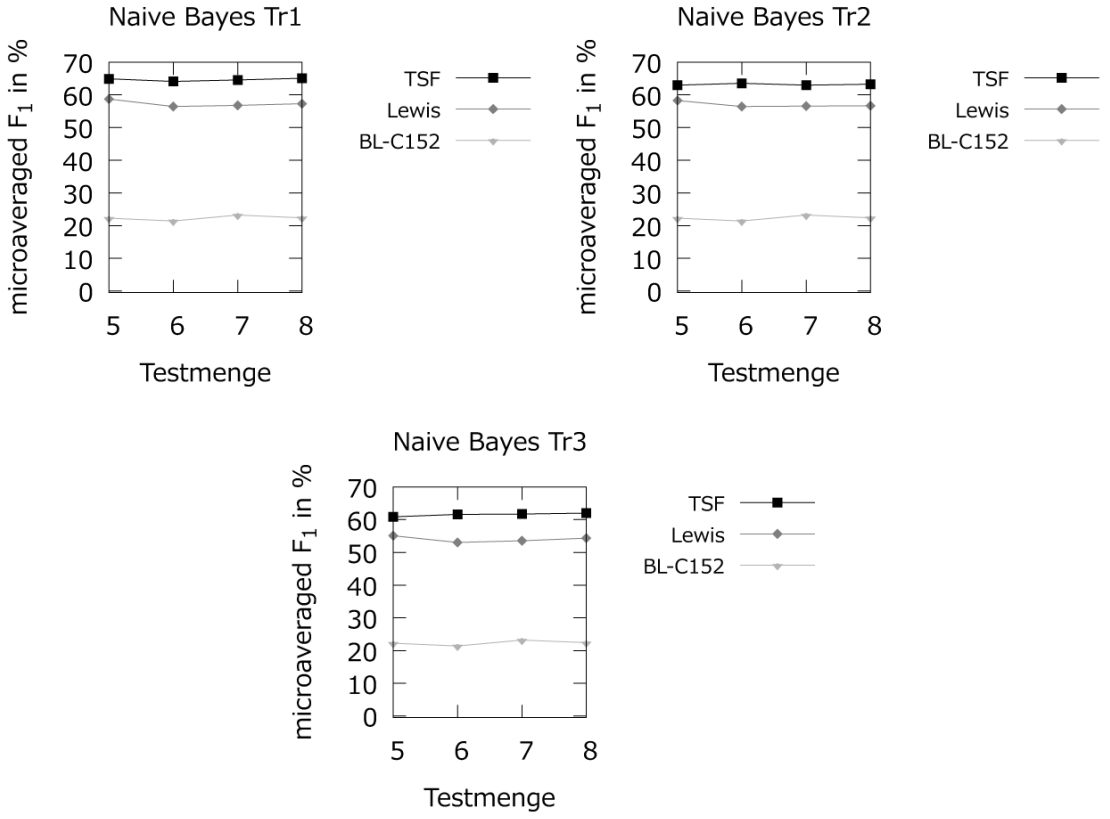


Abbildung 4.96: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den Naive-Bayes-Klassifizierer für das Experiment Klass2TV

Der erste Klassifizierer dieses Experiments erreicht bei der Auswertung des F_1-Maßes mit Schwerpunkt auf Klassen mit vielen Dokumenten für die wortübergreifenden Features bessere Klassifizierungsergebnisse als für einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt einen Abstand von 7,04 Prozentpunkten zwischen den mit den TSF-Features und mit den Einzelwort-Features erreichten Ergebnis-

1 Die Abkürzungen in den Diagrammen bedeuten für dieses Experiment Folgendes: Baseline wird mit „BL“ abgekürzt und „C152“ steht für den Klassennamen der Klasse mit den meisten Trainingsdokumenten.

sen. Der minimale Abstand beträgt 4,73 Prozentpunkte und der maximale Abstand 8,56 Prozentpunkte.

Liegt der Schwerpunkt bei der Auswertung des F_1-Maßes auf Klassen mit wenigen Dokumenten, so ist das Ergebnis gemischt. Für die Trainingsmenge Tr1 erreicht das einzelwortbasierte Verfahren etwas bessere Ergebnisse als das wortübergreifende. Bei den anderen beiden Trainingsmengen ist es genau andersherum. Im Durchschnitt bedeutet das, dass sich die beiden Verfahren um 0,48 Prozentpunkte voneinander unterscheiden. Der minimale Abstand beträgt 0,09 Prozentpunkte und der maximale Abstand 1,81 Prozentpunkte. Da in zwei von drei Fällen die wortübergreifenden Features etwas bessere Klassifizierungsergebnisse im Macroaveraged-Fall für den Naive-Bayes-Klassifizierer erreichen, wird in der Gesamtauswertung ein besseres Ergebnis für die wortübergreifenden Features gezählt.

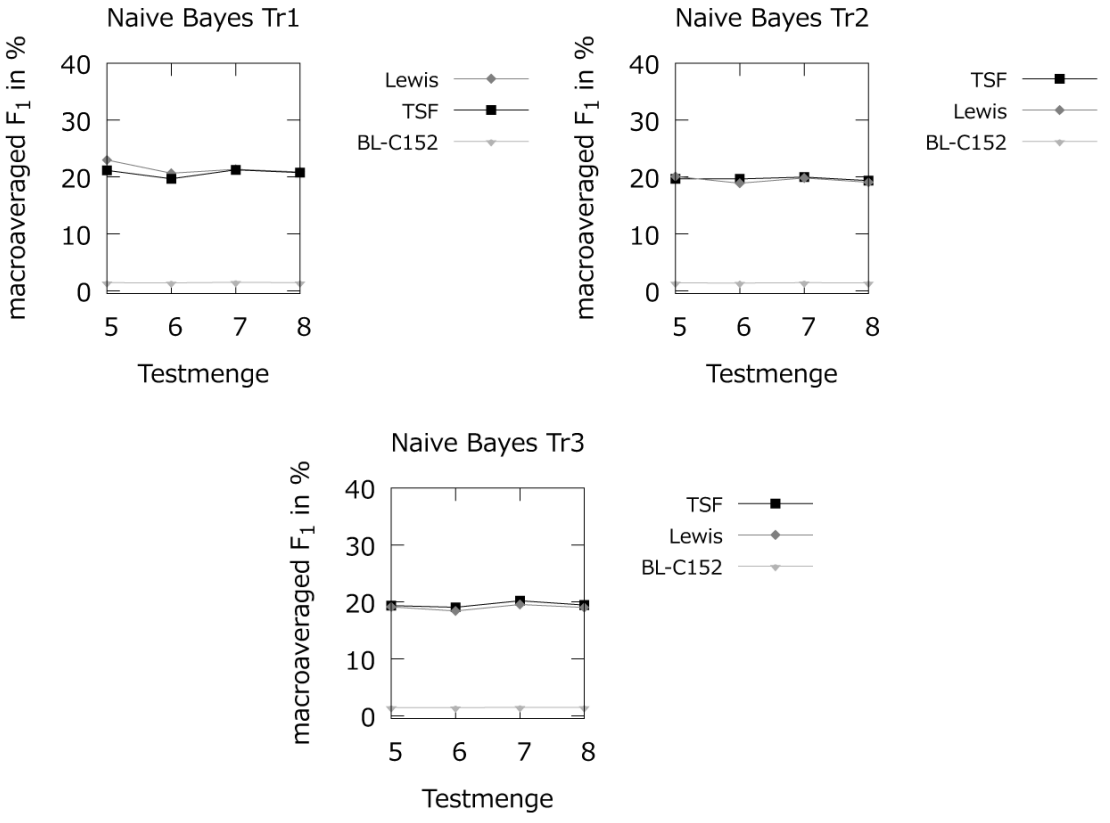


Abbildung 4.97: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den Naive-Bayes-Klassifizierer für das Experiment Klass2TV

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal ca. 65% im Microaveraged-Fall erreicht wird. Im Macroaveraged-Fall sind es ca. 20%. Für beide Verfahren gilt, dass über alle Trainingsteilmengen und beide Auswertungsmöglichkeiten des F_1-Maßes wesentlich bessere Ergebnisse erzielt werden als mit einer zufälligen Zuordnung der Testdokumente zu den zuordenbaren Klassen, d.h., es werden bessere Ergebnisse erreicht als mit der Baseline-Klassifizierung.

– Ergebnisvergleich Decision Tree
  Bildet man die Ergebnisse, die mit dem Decision-Tree-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man die Abbildungen 4.98 und 4.99 auf S. 420 und 421.

  Das Ergebnis ähnelt dem Ergebnis des Experiments Klass1TV insofern, als in beiden Fällen das einzelwortbasierte Verfahren besser abschneidet als das wortübergreifende Verfahren. Im Microaveraged-Fall ist das einzelwortbasierte im Durchschnitt um 4,37 Prozentpunkte besser als das wortübergreifende Verfahren. Der minimale Abstand beträgt 3,03 Prozentpunkte und der maximale Abstand 5,81 Prozentpunkte. Auch im Macroaveraged-Fall erreicht das einzelwortbasierte Verfahren für den Decision-Tree-Klassifizierer im Durchschnitt um 2,83 Prozentpunkte bessere Ergebnisse als das wortübergreifende Verfahren. Hier beträgt der maximale Abstand 4,08 Prozentpunkte und der minimale Abstand 1,38 Prozentpunkte. Eine mögliche Erklärung für das schlechtere Abschneiden der wortübergreifenden Features wurde bereits beim Experiment Klass1TV genannt. Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal ca. 57% für den Fall der Auswertung des F_1-Maßes mit Schwerpunkt auf Klassen mit vielen Dokumenten erreicht wird.

Für den Fall der Auswertung des F_1-Maßes mit Schwerpunkt auf Klassen mit wenigen Dokumenten sind es ca. 23%. Für beide Verfahren gilt, dass über alle Trainingsteilmengen bei beiden Auswertungsmöglichkeiten des F_1-Maßes wesentlich bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung.

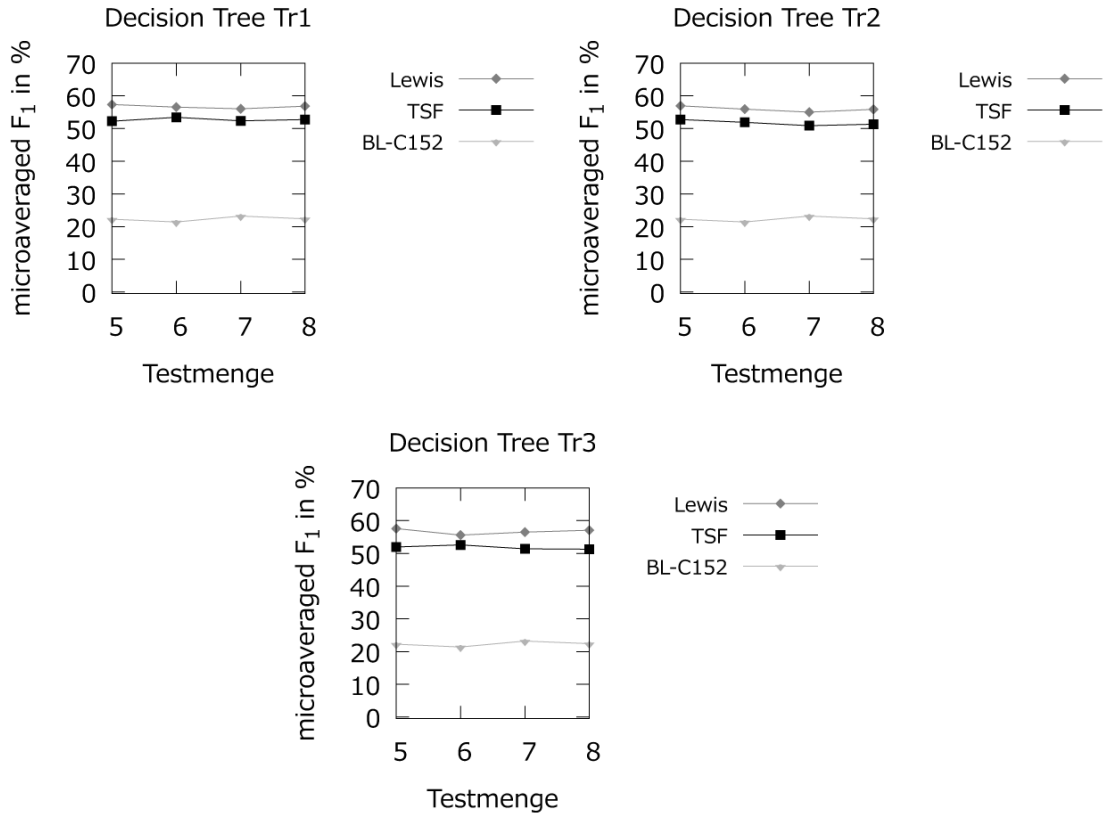


Abbildung 4.98: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den Decision-Tree-Klassifizierer für das Experiment Klass2TV

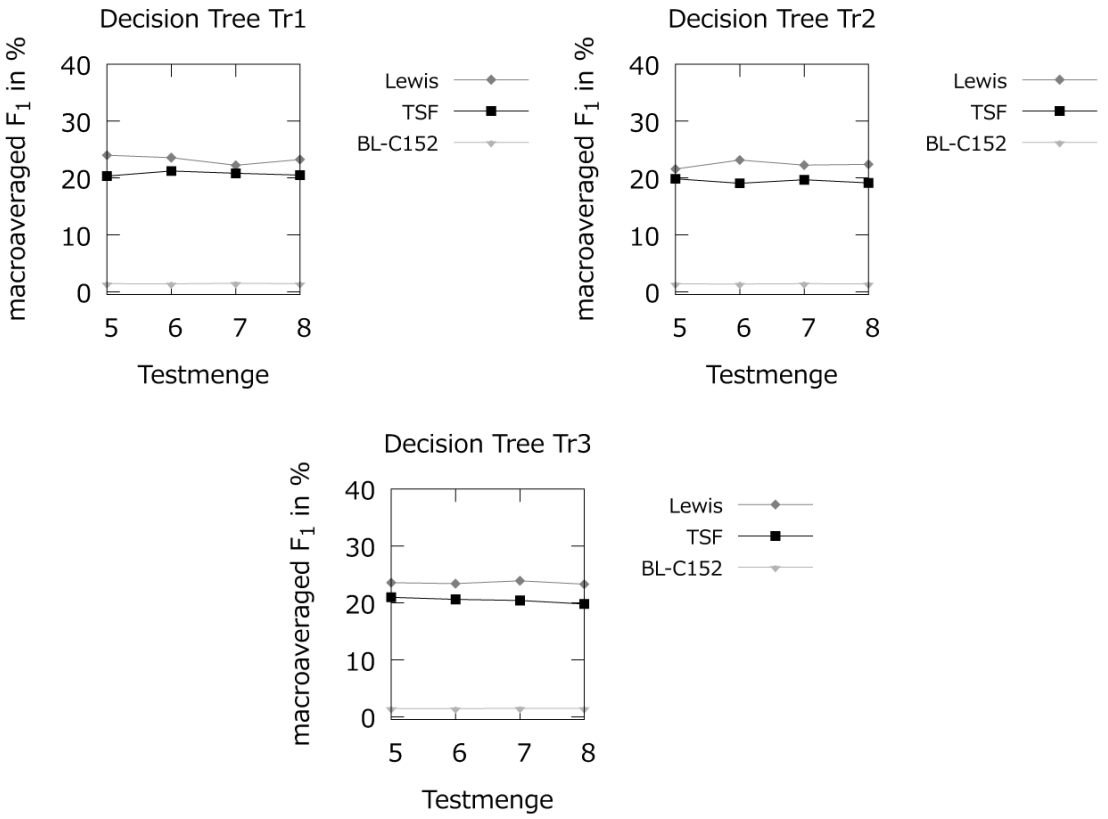


Abbildung 4.99: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den Decision-Tree-Klassifizierer für das Experiment Klass2TV

– Ergebnisvergleich k-Nearest-Neighbour
  Bildet man die Ergebnisse, die mit dem 10-Nearest-Neighbour-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man die Abbildungen 4.100 und 4.101 auf S. 422 und 423.

  Die abgebildeten Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen für den Fall, dass bei der Auswertung des F_1-Maßes der Schwerpunkt auf Klassen mit vielen Dokumenten liegt, so erhält man im Durchschnitt einen Abstand von 43,18 Prozentpunkten zwischen den mit den TSF-Features und mit den Einzelwort-Features erreichten Ergebnissen. Das ist der bisher deutlichste gemessene Abstand in diesem Experiment.

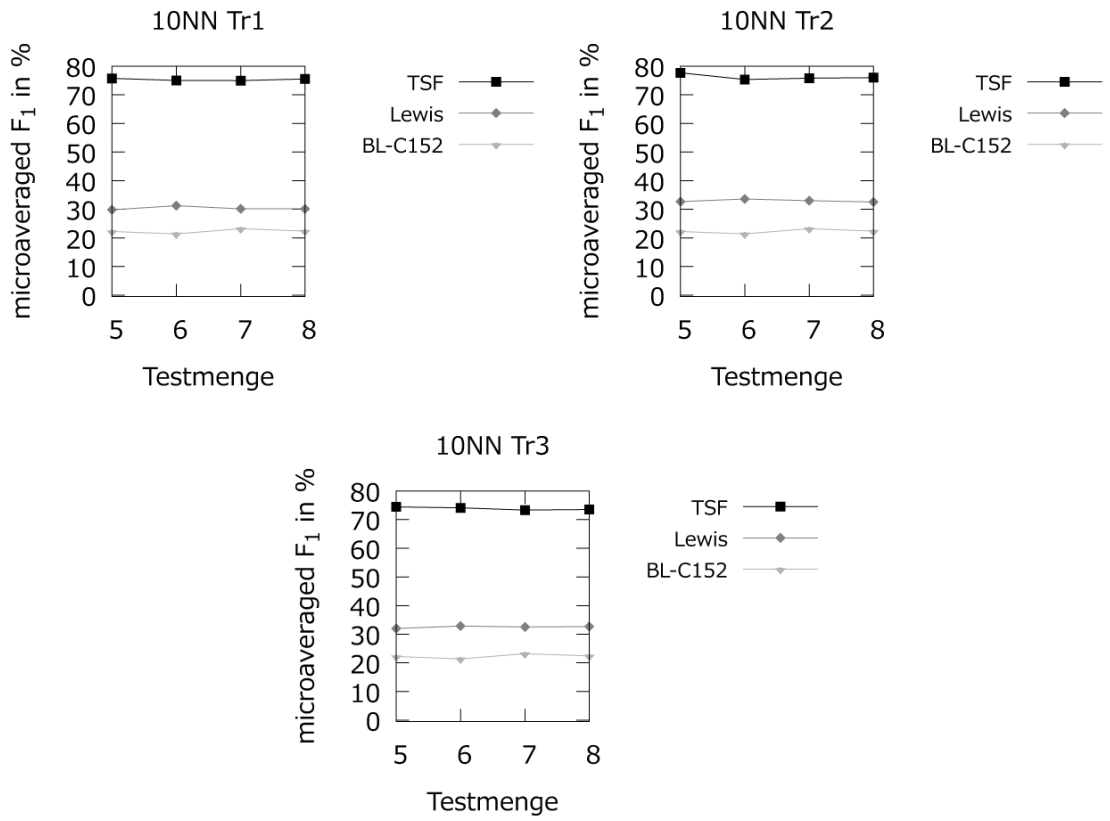


Abbildung 4.100: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den 10-Nearest-Neighbour-Klassifizierer für das Experiment Klass2TV

Der minimale Abstand beträgt 40,75 Prozentpunkte und der maximale Abstand 45,92 Prozentpunkte. Beim 10NN-Klassifizierer liegen die Ergebnisse beider Verfahren für diesen Fall wieder weit auseinander. Wird der Schwerpunkt beim Auswerten des F_1-Maßes auf Klassen mit wenigen Dokumenten gelegt, so ergibt sich ein ähnliches Bild. Die Ergebnisse mit den TSF-Feature-Vektoren übertreffen diejenigen mit den einzelwortbasierten Feature-Vektoren im Durchschnitt um 22,58 Prozentpunkte. Der maximale Abstand zwischen den Ergebnissen beider Verfahren liegt bei 23,71 Prozentpunkten und der minimale Abstand bei 20,40 Prozentpunkten.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch.

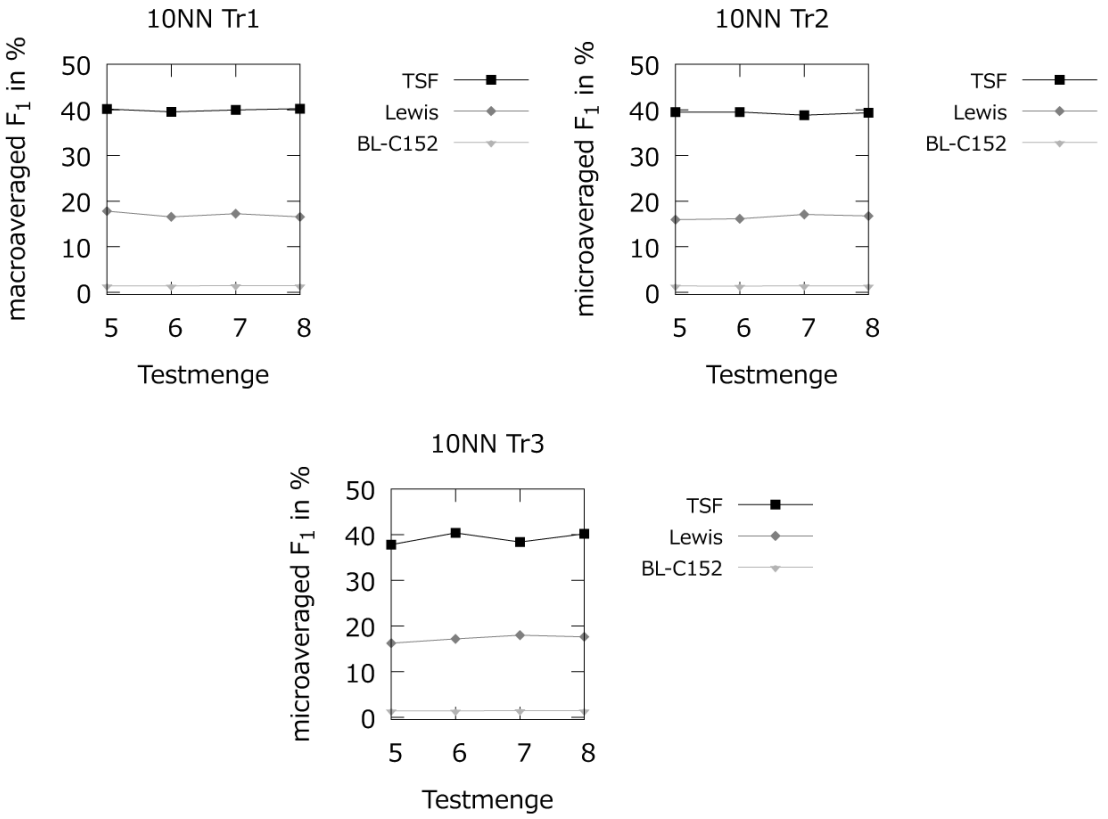


Abbildung 4.101: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den 10-Nearest-Neighbour-Klassifizierer für das Experiment Klass2TV

Die Klassifizierungsergebnisse zeigen, dass eine Qualität von ca. 75% im Microaveraged-Fall erreicht wird. Im Macroaveraged-Fall werden für den 10NN-Klassifizierer um die 40% erreicht. Für beide Verfahren und beide Auswertungsmöglichkeiten des F_1-Maßes gilt, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung.

Bildet man die Ergebnisse, die mit dem 20-Nearest-Neighbour-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man die Abbildungen 4.102 und 4.103 auf S. 424 und 425.

Auch diese Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen für den Microaveraged-Fall, so erhält man im Durchschnitt einen

Abstand von 42,37 Prozentpunkten zwischen den erreichten Ergebnissen mit den TSF-Features und den Einzelwort-Features. Der minimale Abstand beträgt 40,37 Prozentpunkte und der maximale Abstand 44,89 Prozentpunkte. Im Macroaveraged-Fall erreicht der 20NN-Klassifizierer basierend auf wortübergreifenden Features ebenfalls bessere Ergebnisse für alle drei Trainingsteilmengen als derjenige, der auf Einzelworten als Features basiert. Der minimale Abstand zwischen den erreichten Ergebnissen beträgt hier 18,46 Prozentpunkte und der maximale Abstand 22,23 Prozentpunkte. Das ergibt im Durchschnitt einen Abstand von 20,47 Prozentpunkten.

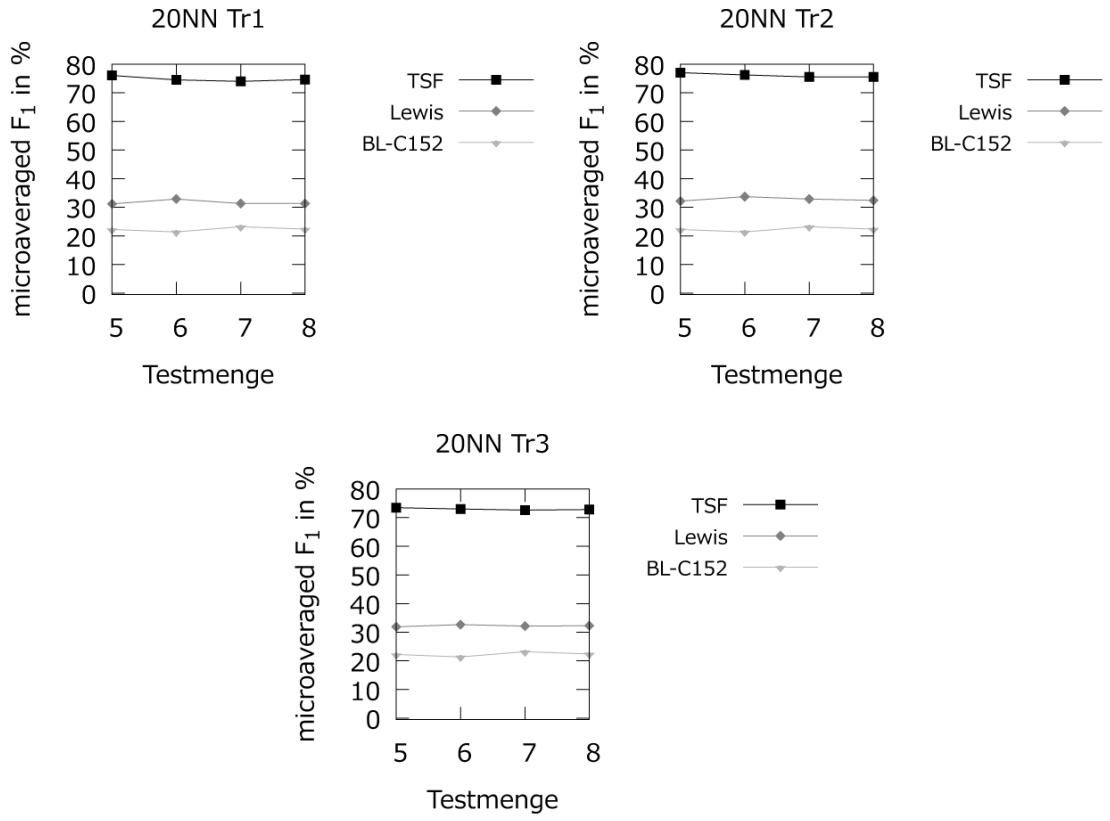


Abbildung 4.102: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den 20-Nearest-Neighbour-Klassifizierer für das Experiment Klass2TV

Die absoluten Klassifizierungsergebnisse stehen in der vorliegenden Arbeit nicht im Vordergrund, da es um den Vergleich zwischen den beiden Verfahren zur Textaufbereitung für die Klassifizierung geht. Ein paar Anmerkungen zum absoluten Ergebnis folgen hier dennoch.

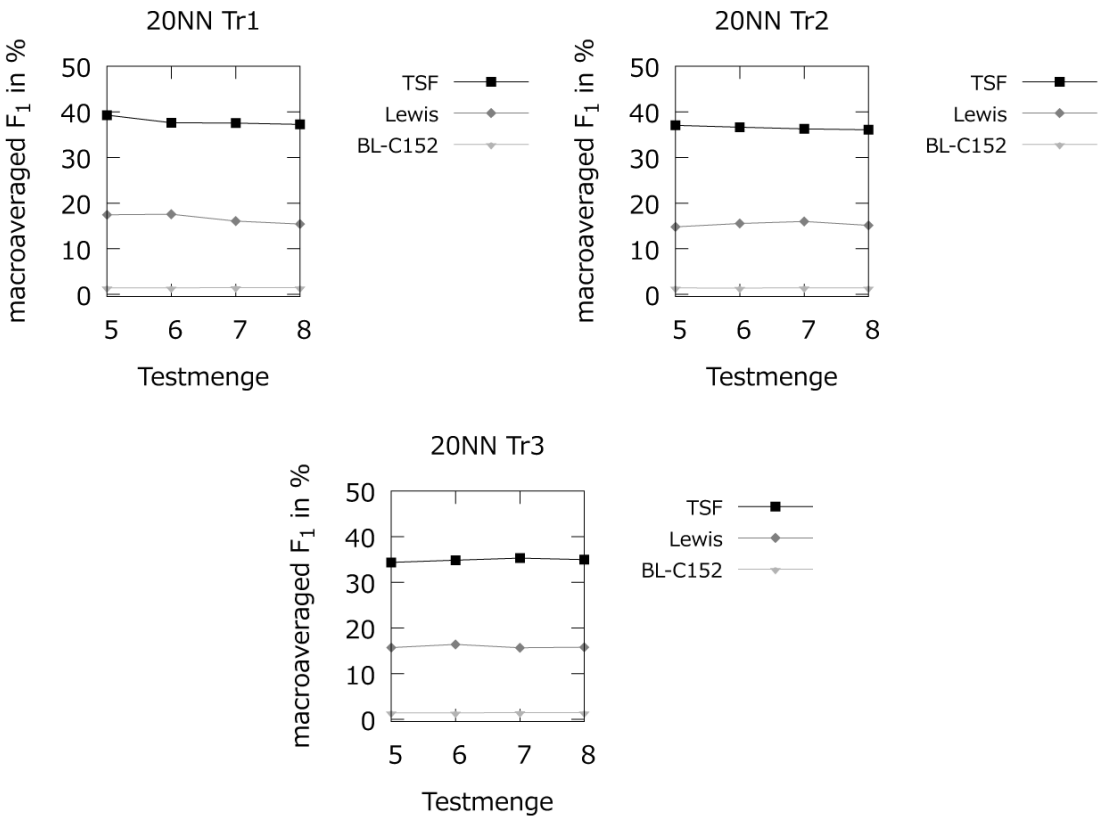


Abbildung 4.103: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den 20-Nearest-Neighbour-Klassifizierer für das Experiment Klass2TV

Die Klassifizierungsergebnisse zeigen, dass eine Qualität von ca. 75% erreicht wird, wenn der Schwerpunkt bei der Auswertung des F_1-Maßes auf Klassen mit vielen Dokumenten liegt. Liegt der Schwerpunkt der Auswertung auf Klassen mit wenigen Dokumenten, wird eine Qualität von ca. 40% erreicht. Für beide Verfahren gilt für beide Auswertungsmöglichkeiten des F_1-Maßes, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung.

Bildet man die Ergebnisse, die mit dem 200-Nearest-Neighbour-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man die Abbildungen 4.104 und 4.105 auf S. 426 und 427.

Diese Ergebnisse stützen ebenfalls die der Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwi-

schen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt für den Microaveraged-Fall einen Abstand von 28,37 Prozentpunkten zwischen den mit den TSF-Features und mit den Einzelwort-Features erreichten Ergebnissen. Der minimale Abstand beträgt 25,2 Prozentpunkte und der maximale Abstand 30,44 Prozentpunkte. Die Ergebnisse für die Auswertung des F_1-Maßes mit Schwerpunkt auf Klassen mit wenigen Dokumenten zeigen, dass der Klassifizierer mit TSF-Feature-Vektoren im Durchschnitt um 10,63 Prozentpunkte bessere Ergebnisse liefert als der gleiche Klassifizierer mit einzelwortbasierten Features. Der minimale Abstand beträgt in diesem Fall 9,06 Prozentpunkte und der maximale Abstand 11,81 Prozentpunkte.

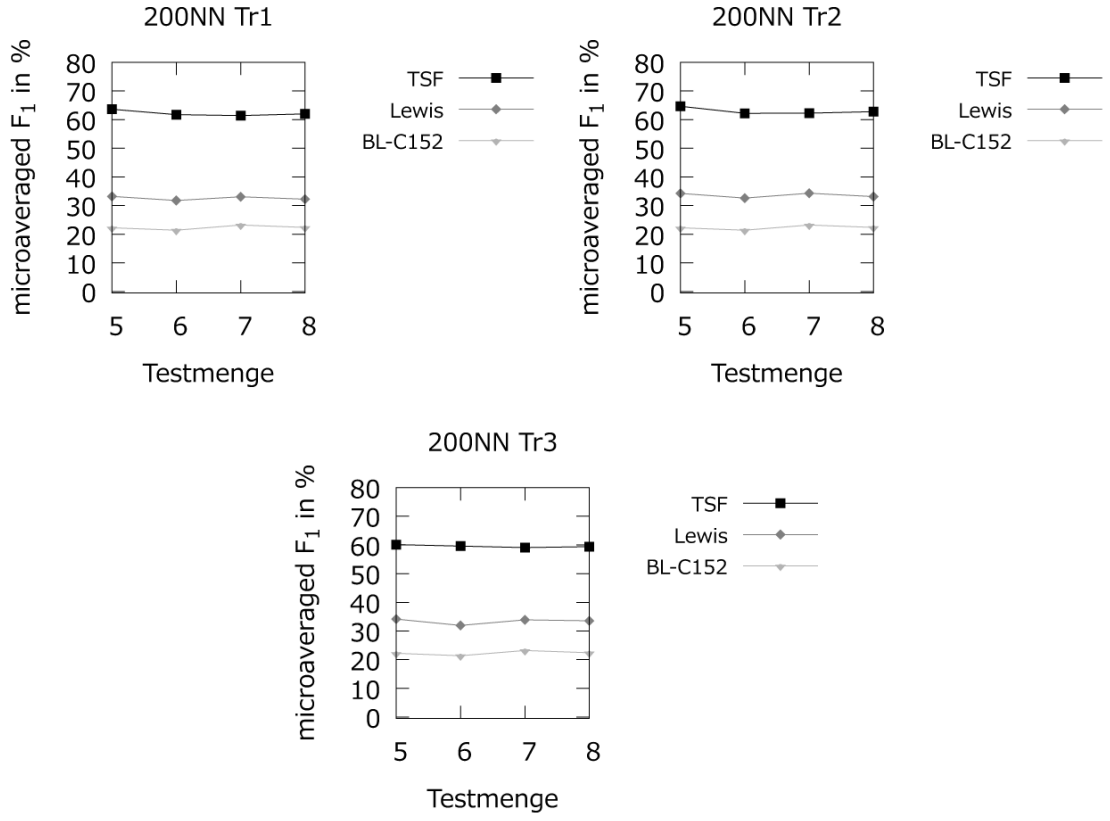


Abbildung 4.104: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den 200-Nearest-Neighbour-Klassifizierer für das Experiment Klass2TV

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmer-

kungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass im Microaveraged-Fall eine Qualität von ca. 63% richtig klassifizierter Dokumente erreicht wird.

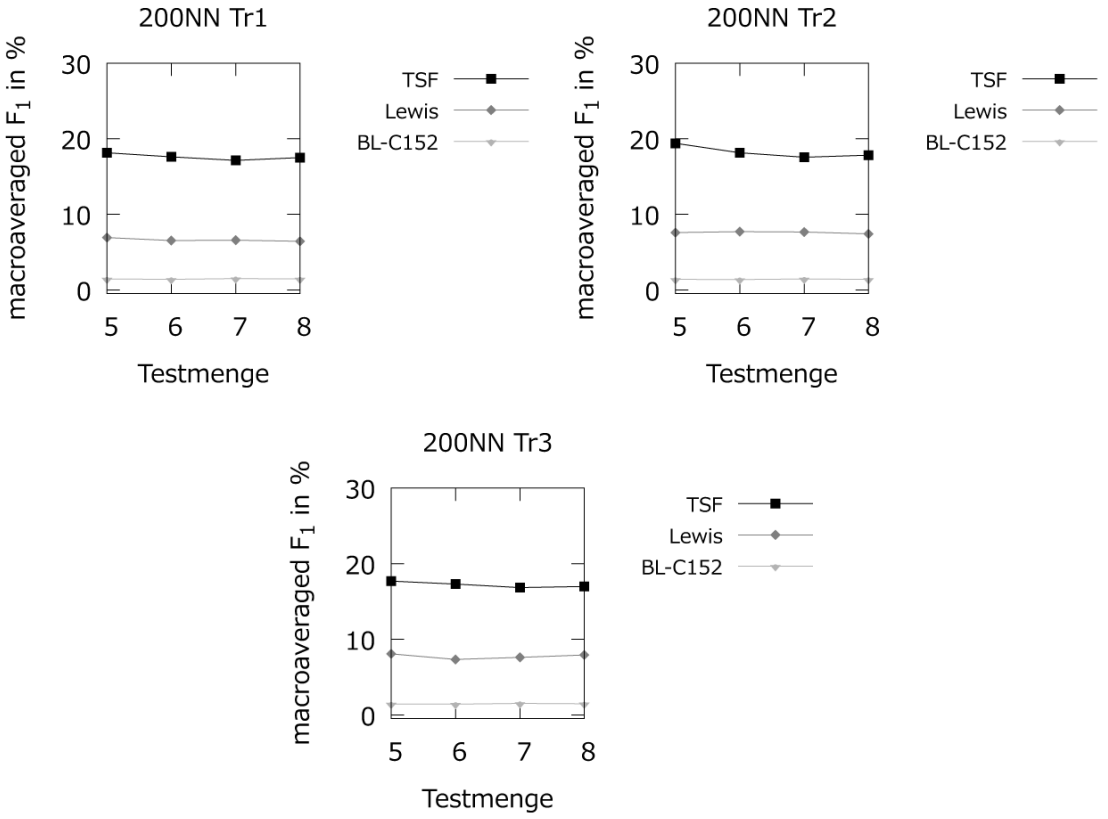


Abbildung 4.105: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den 200-Nearest-Neighbour-Klassifizierer für das Experiment Klass2TV

Im Macroaveraged-Fall ist es eine Qualität von ca. 18%. Für beide Verfahren gilt für beide Auswertungsmöglichkeiten des F_1-Maßes, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung.

Insgesamt ergibt sich für den k-Nearest-Neighbour-Klassifizierer folgendes Bild:

* Die TSF-Feature-Vektoren erreichen in allen Fällen bessere Ergebnisse als die einzelwortbasierten Feature-Vektoren.
* Die Anzahl der Nachbarn spielt eine entscheidende Rolle bei den erreich-

ten Ergebnissen und dem Abstand zwischen den TSF-Feature-Vektoren und den einzelwortbasierten Feature-Vektoren.
Die optimale Anzahl von Nachbarn scheint für das durchgeführte Experiment in der Nähe von 10 zu liegen, da die Ergebnisse mit 20 Nachbarn ein wenig schlechter und mit 200 Nachbarn noch schlechter sind. Das gilt für beide Verfahren. Sehr deutlich sieht man das am wesentlich schlechteren Ergebnis für die Macroaveraged-Auswertung bei 200 Nachbarn.

– Ergebnisvergleich Support Vector Machine
Bildet man die Ergebnisse, die mit dem Support-Vector-Machine-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden in Diagrammen ab, so erhält man die Abbildungen 4.106 und 4.107 auf S. 429 und 430.

Die Ergebnisse der Support Vector Machine zeigen, dass im Microaveraged-Fall die SVM basierend auf TSF-Feature-Vektoren im Durchschnitt um 10,53 Prozentpunkte besser abschneidet als diejenige basierend auf einzelwortbasierten Feature-Vektoren. Als maximaler Abstand für die Support Vector Machine ergibt sich ein Wert von 15,13 Prozentpunkten und als minimaler Abstand ein Wert von 5,28 Prozentpunkten. Im Macroaveraged-Fall ergibt sich ein minimaler Abstand von 3,23 Prozentpunkten und ein maximaler Abstand von 8,35 Prozentpunkten.

Das bedeutet, im Durchschnitt scheidet der auf TSF-Feature-Vektoren basierende SVM-Klassifizierer um 6,06 Prozentpunkte besser ab als der auf Einzelworten basierende SVM-Klassifizierer.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von ca. 76% im Microaveraged-Fall erreicht wird. Im Macroaveraged-Fall sind es ca. 35%. Für beide Verfahren gilt, dass über alle Trainingsteilmengen und beide Auswertungsarten des F_1-Maßes wesentlich bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung.

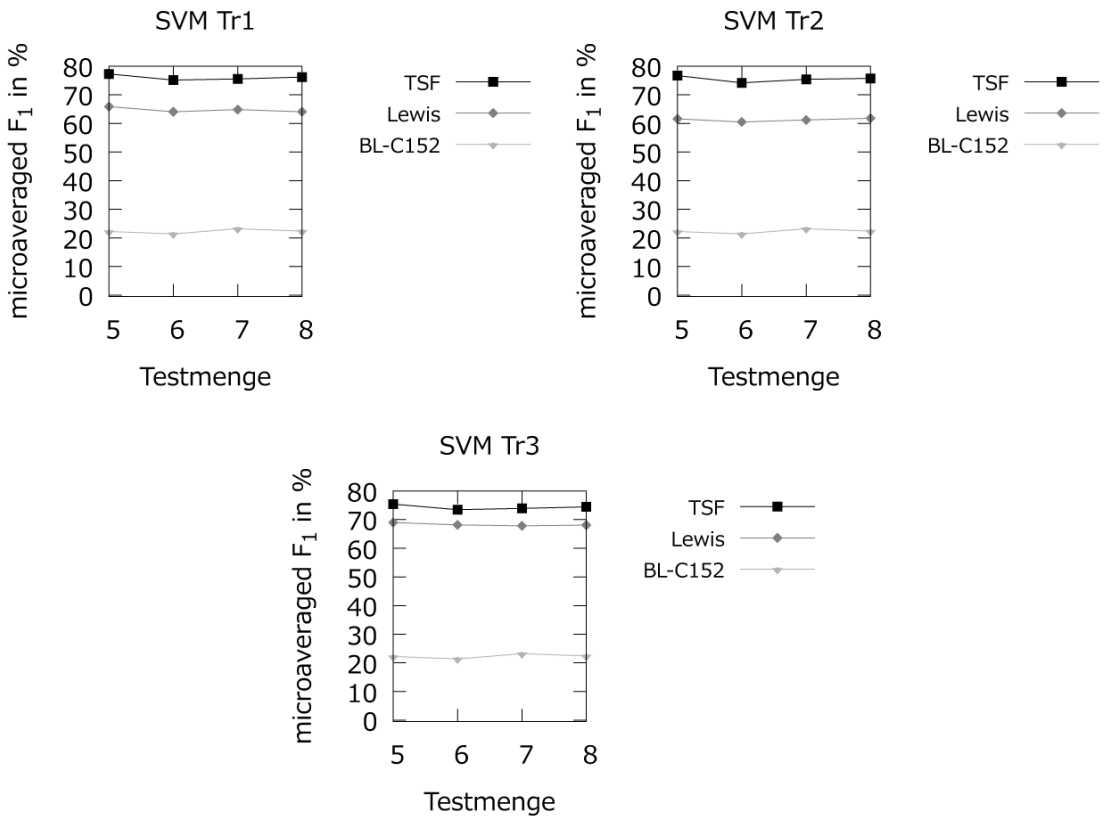


Abbildung 4.106: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den Support-Vector-Machine-Klassifizierer für das Experiment Klass2TV

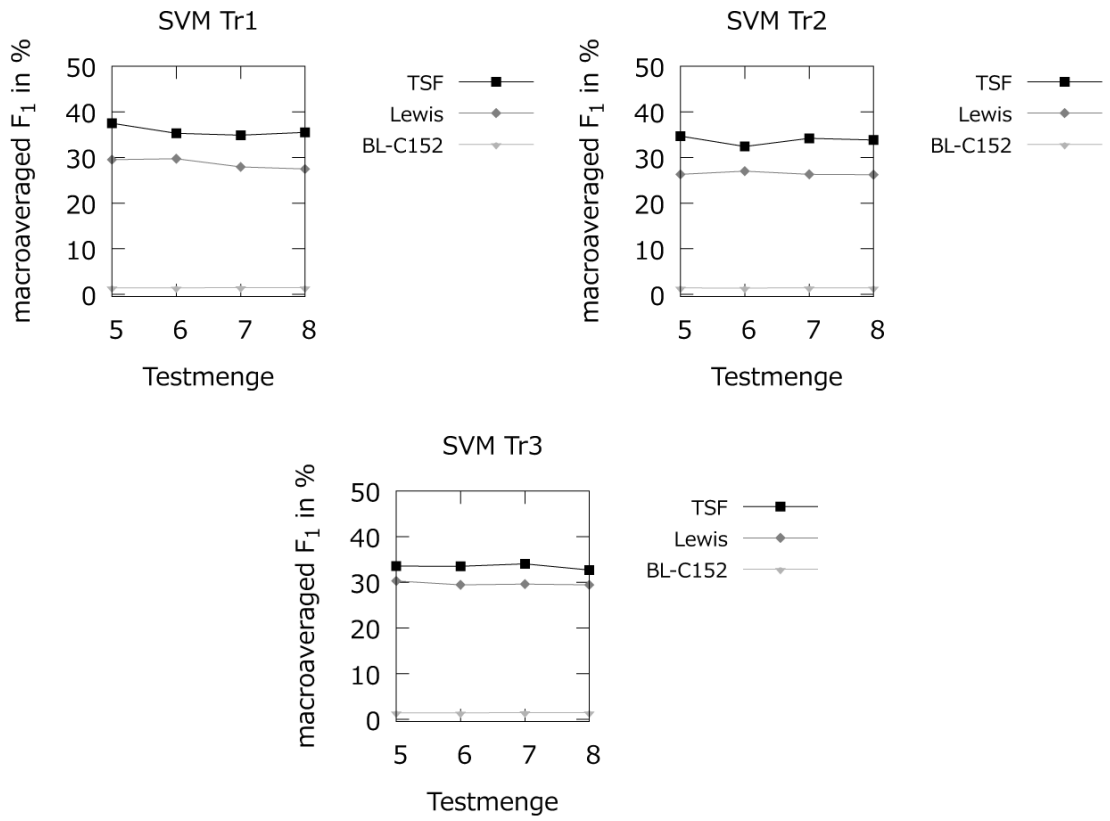


Abbildung 4.107: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den Support-Vector-Machine-Klassifizierer für das Experiment Klass2TV

– Ergebnisvergleich über alle Algorithmen für das Experiment
Vergleicht man die Ergebnisse des einzelwortbasierten Verfahrens und des Verfahrens mit TSF-Features über alle Algorithmen miteinander, so stellt man Folgendes fest: Das auf TSF-Features basierende Verfahren schneidet beim Klassifizieren von natürlichsprachlichen Dokumenten für fast alle verwendeten Algorithmen besser ab als das einzelwortbasierte Verfahren. Der Algorithmus, für den das nicht zutrifft, ist der Decision-Tree-Algorithmus. Eine mögliche Ursache für das schlechtere Abschneiden der TSF-Feature-Vektoren wurde bereits im entsprechenden Unterkapitel genannt.

Als Tabelle zusammengefasst ergibt sich für das fünfte Experiment folgendes Bild[1]:

Tabelle 4.95: Ergebnisse des Experiments Klass2TV

Verfahren	Algorithmus							
	NB (micro)	NB (macro)	DT (micro)	DT (macro)	kNN (micro)	kNN (macro)	SVM (micro)	SVM (macro)
TSF-Feature-Vektoren	✓	✓			✓	✓	✓	✓
Lewis-Feature-Vektoren			✓	✓				

Damit erreichen die TSF-Feature-Vektoren in 10 von 12 Fällen ein besseres Klassifizierungergebnis als die einzelwortbasierten Feature-Vektoren.[2]

Somit stützt dieses fünfte Experiment die Vermutung, dass wortübergreifende Features qualitativ bessere Ergebnisse beim Klassifizieren von natürlichsprachlichen Dokumenten liefern als Verfahren, die natürlichsprachliche Dokumente in Einzelworte aufteilen und diese als Features für das Klassifizieren benutzen.

4.3.2.2.3 Experiment 6

- ID
  Klass2IV
- Bezeichnung
  vektorbasiertes Single-label-Industry-Klassifizieren von Testdaten der Reuters-Daten RCV1-v2

1 Ein Häkchen zeigt dabei das Verfahren an, das für den betreffenden Algorithmus die höchste Qualität über alle Trainings- und Testteilmengen erreicht hat.
2 Hierbei werden die unterschiedlichen Anzahlen an Nachbarn beim kNN einzeln gezählt.

- Trainingsdaten
  - 3 randomisiert zusammengestellte Teilmengen der Trainingsdaten des RCV1-v2 mit jeweils 2.000 Dokumenten: Tr1, Tr2, Tr3[1]
  - Dokumente von Tr1 bis Tr3 gehören zu genau einer Industry-Klasse
- Testdaten
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 2.000 Dokumenten: Te5, Te6[2]
  - Dokumente in Te5 und Te6 gehören zu genau einer Industry-Klasse
  - 2 randomisiert zusammengestellte Teilmengen der Testdaten des RCV1-v2 mit jeweils 10.000 Dokumenten: Te7, Te8
  - Dokumente in Te7 und Te8 gehören zu genau einer Industry-Klasse
- Ähnlichkeit
  vektorbasiert
- Algorithmen
  - Naive Bayes
  - Decision Tree
  - k-Nearest-Neighbour mit $k = 10$, 20 und 200 Nachbarn mit Cosinus zur Ähnlichkeitsberechnung zwischen den Vektoren
  - Support Vector Machine SVM^{multiclass} mit Parameter $C = 1,0$ und linearem Kernel
- Baseline
  - randomisierte Zuordnung
  - Majority-Class-Zuordnung zu den zwei Klassen mit den meisten Trainingsdokumenten
- Evaluation
  F_1-Maß in der micro- und macroaveraged Form

1 Die Trainingsdaten entsprechen den Trainingsdaten des Experiments Klass1IV.
2 Die Testdaten entsprechen den Testdaten des Experiments Klass1IV. Daher beginnt die Numerierung nicht bei Te1.

- Ergebnisse[1]
  Für das Klassifizieren der Testmengen Te5 bis Te8 mit dem Naive-Bayes-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.96, 4.97 und 4.98, zu sehen auf S. 433 bis 434.
  Für das Klassifizieren der Testmengen Te5 bis Te8 mit dem Decision-Tree-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.99, 4.100 und 4.101, zu sehen auf S. 434 bis 435.
  Für das Klassifizieren der Testmengen Te5 bis Te8 mit dem k-Nearest-Neighbour-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3 und 10, 20 und 200 Nachbarn, ergeben sich die Ergebnisse aus den Tabellen 4.102, 4.103 4.104, 4.105, 4.106, 4.107, 4.108, 4.109 und 4.110, zu sehen auf S. 436 bis 439.
  Für das Klassifizieren der Testmengen Te5 bis Te8 mit dem Support-Vector-Machine-Klassifizierer, trainiert mit den Trainingsmengen Tr1 bis Tr3, ergeben sich die Ergebnisse aus den Tabellen 4.111, 4.112 und 4.113, zu sehen auf S. 439 bis 440.

Tabelle 4.96: Ergebnisse des Experiments Klass2IV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	807	4615	3103	17,30	3,43
	Industry	6	2000	849	4576	3064	18,18	3,09
	Industry	7	10000	4372	23076	15240	18,58	3,59
	Industry	8	10000	4215	23073	15250	18,03	3,55
Lewis-Feature-Vektoren	Industry	5	2000	920	3870	2965	21,21	4,48
	Industry	6	2000	951	3819	2941	21,96	4,40
	Industry	7	10000	4921	19018	14572	22,66	4,72
	Industry	8	10000	4760	19083	14619	22,03	4,64

1 In allen Tabellen werden die Ergebnisse auf zwei Stellen nach dem Komma gerundet. Alle Klassifizierer sind im abschließenden Vergleich durch ihre Abkürzungen aufgelistet: NB für Naive Bayes, DT für Decision Tree, kNN für k-Nearest-Neighbour und SVM für Support Vector Machine.

Tabelle 4.97: Ergebnisse des Experiments Klass2IV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	739	4610	3199	15,91	3,16
	Industry	6	2000	805	4581	3133	17,27	3,63
	Industry	7	10000	4157	22967	15567	17,75	3,53
	Industry	8	10000	4021	23020	15531	17,26	3,59
Lewis-Feature-Vektoren	Industry	5	2000	904	4081	3005	20,33	5,15
	Industry	6	2000	920	4093	2988	20,63	5,00
	Industry	7	10000	4795	20288	14767	21,48	4,88
	Industry	8	10000	4636	20301	14769	20,91	5,04

Tabelle 4.98: Ergebnisse des Experiments Klass2IV für den Naive-Bayes-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	845	4106	3066	19,07	3,93
	Industry	6	2000	887	4114	3023	19,91	3,16
	Industry	7	10000	4388	20994	15243	19,50	3,46
	Industry	8	10000	4281	20881	15205	19,18	3,72
Lewis-Feature-Vektoren	Industry	5	2000	873	3686	3036	20,62	3,99
	Industry	6	2000	915	3686	3000	21,49	4,19
	Industry	7	10000	4737	18540	14820	22,12	5,09
	Industry	8	10000	4608	18452	14843	21,68	4,73

Tabelle 4.99: Ergebnisse des Experiments Klass2IV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	539	3041	3481	14,18	2,98
	Industry	6	2000	557	3043	3444	14,66	2,87
	Industry	7	10000	2785	15189	17274	14,65	3,18
	Industry	8	10000	2874	15060	17013	15,20	3,54
Lewis-	Industry	5	2000	693	2947	3261	18,25	4,30
wird auf der nächsten Seite fortgesetzt								

Tabelle 4.99: Ergebnisse des Experiments Klass2IV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr1 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
Feature-Vektoren	Industry	6	2000	672	2963	3265	17,75	4,10
	Industry	7	10000	3404	14782	16421	17,91	4,48
	Industry	8	10000	3276	14930	16406	17,29	4,28

Tabelle 4.100: Ergebnisse des Experiments Klass2IV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	605	2964	3370	16,04	3,92
	Industry	6	2000	617	2966	3337	16,37	3,88
	Industry	7	10000	3185	14696	16731	16,85	4,16
	Industry	8	10000	3057	14725	16692	16,29	3,86
Lewis-Feature-Vektoren	Industry	5	2000	650	3091	3316	16,87	4,05
	Industry	6	2000	688	3136	3273	17,68	4,08
	Industry	7	10000	3602	15321	16221	18,59	4,64
	Industry	8	10000	3330	15486	16393	17,28	3,87

Tabelle 4.101: Ergebnisse des Experiments Klass2IV für den Decision-Tree-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	618	3020	3373	16,20	3,83
	Industry	6	2000	633	3080	3351	16,45	3,91
	Industry	7	10000	3220	15301	16798	16,71	3,68
	Industry	8	10000	3020	15457	16839	15,76	3,37
Lewis-Feature-Vektoren	Industry	5	2000	708	2893	3236	18,77	4,74
	Industry	6	2000	648	3042	3317	16,93	3,65
	Industry	7	10000	3593	14820	16224	18,80	4,31
	Industry	8	10000	3502	14915	16163	18,39	4,48

Tabelle 4.102: Ergebnisse des Experiments Klass2IV für den 10NN-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	907	2510	2927	25,02	5,93
	Industry	6	2000	984	2398	2827	27,36	6,25
	Industry	7	10000	4867	12229	14283	26,86	6,26
	Industry	8	10000	4849	12085	14233	26,93	6,11
Lewis-Feature-Vektoren	Industry	5	2000	344	3474	3660	8,80	2,46
	Industry	6	2000	384	3450	3608	9,81	2,50
	Industry	7	10000	1861	17372	18131	9,49	2,80
	Industry	8	10000	1770	17461	18077	9,06	2,55

Tabelle 4.103: Ergebnisse des Experiments Klass2IV für den 10NN-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	953	2423	2866	26,49	5,88
	Industry	6	2000	922	2424	2889	25,76	5,11
	Industry	7	10000	4692	11907	14458	26,25	5,69
	Industry	8	10000	4742	11977	14329	26,50	5,85
Lewis-Feature-Vektoren	Industry	5	2000	337	3475	3679	8,61	3,07
	Industry	6	2000	339	3487	3648	8,68	2,68
	Industry	7	10000	1699	17480	18360	8,66	2,72
	Industry	8	10000	1734	17536	18183	8,85	2,79

Tabelle 4.104: Ergebnisse des Experiments Klass2IV für den 10NN-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	917	2415	2926	25,56	6,09
	Industry	6	2000	973	2398	2858	27,02	5,80
	Industry	7	10000	4738	12054	14484	26,31	5,90
	Industry	8	10000	4724	12073	14387	26,31	6,14
Lewis-	Industry	5	2000	334	3421	3686	8,59	2,02

wird auf der nächsten Seite fortgesetzt

Tabelle 4.104: Ergebnisse des Experiments Klass2IV für den 10NN-Klassifizierer für Trainingsmenge Tr3 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
Feature-Vektoren	Industry	6	2000	366	3432	3638	9,38	2,64
	Industry	7	10000	1694	17301	18387	8,67	2,58
	Industry	8	10000	1747	17245	18200	8,97	2,75

Tabelle 4.105: Ergebnisse des Experiments Klass2IV für den 20NN-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	1012	2419	2801	27,94	5,91
	Industry	6	2000	1052	2373	2756	29,09	5,49
	Industry	7	10000	5160	11922	13957	28,51	5,47
	Industry	8	10000	5062	11874	13958	28,16	5,33
Lewis-Feature-Vektoren	Industry	5	2000	359	3435	3633	9,22	2,40
	Industry	6	2000	381	3430	3609	9,77	2,23
	Industry	7	10000	1861	17316	18112	9,51	2,27
	Industry	8	10000	1762	17412	18078	9,03	2,03

Tabelle 4.106: Ergebnisse des Experiments Klass2IV für den 20NN-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	985	2471	2810	27,17	5,58
	Industry	6	2000	976	2423	2818	27,14	4,77
	Industry	7	10000	5092	11854	13957	28,29	5,51
	Industry	8	10000	5048	12002	13957	28,00	5,43
Lewis-Feature-Vektoren	Industry	5	2000	307	2172	3702	9,46	3,20
	Industry	6	2000	306	2151	3688	9,49	2,36
	Industry	7	10000	1566	10727	18477	9,69	2,28
	Industry	8	10000	1548	10719	18361	9,62	2,21

Tabelle 4.107: Ergebnisse des Experiments Klass2IV für den 20NN-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	957	2356	2860	26,84	5,48
	Industry	6	2000	1018	2341	2794	28,39	5,44
	Industry	7	10000	5071	11683	14050	28,27	5,49
	Industry	8	10000	4994	11563	14031	28,07	5,78
Lewis-Feature-Vektoren	Industry	5	2000	341	3418	3663	8,79	1,83
	Industry	6	2000	418	3377	3576	10,73	2,62
	Industry	7	10000	1886	17071	18085	9,69	2,47
	Industry	8	10000	1757	17200	18105	9,05	2,41

Tabelle 4.108: Ergebnisse des Experiments Klass2IV für den 200NN-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	1044	2950	2772	26,73	3,49
	Industry	6	2000	1102	2909	2705	28,19	3,68
	Industry	7	10000	5553	14582	13515	28,33	3,53
	Industry	8	10000	5384	14665	13598	27,59	3,48
Lewis-Feature-Vektoren	Industry	5	2000	628	4786	3347	13,38	1,09
	Industry	6	2000	700	4748	3269	14,87	1,26
	Industry	7	10000	3603	23588	16267	15,31	1,16
	Industry	8	10000	3471	23761	16239	14,79	1,25

Tabelle 4.109: Ergebnisse des Experiments Klass2IV für den 200NN-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	1040	3161	2768	25,97	3,30
	Industry	6	2000	1071	3171	2735	26,62	3,30
	Industry	7	10000	5434	16029	13653	26,80	3,26
	Industry	8	10000	5283	15956	13703	26,27	3,16
Lewis-	Industry	5	2000	625	4924	3358	13,11	1,25

wird auf der nächsten Seite fortgesetzt

Tabelle 4.109: Ergebnisse des Experiments Klass2IV für den 200NN-Klassifizierer für Trainingsmenge Tr2 (Fortsetzung)

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
Feature-Vektoren	Industry	6	2000	693	4892	3271	14,51	1,16
	Industry	7	10000	3610	24464	16290	15,05	1,23
	Industry	8	10000	3430	24734	16304	14,32	1,09

Tabelle 4.110: Ergebnisse des Experiments Klass2IV für den 200NN-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	978	2957	2846	25,21	3,05
	Industry	6	2000	1033	2950	2781	26,50	3,28
	Industry	7	10000	5132	14800	14005	26,27	2,95
	Industry	8	10000	4935	14823	14104	25,44	2,87
Lewis-Feature-Vektoren	Industry	5	2000	621	4717	3362	13,32	1,19
	Industry	6	2000	693	4708	3287	14,77	1,14
	Industry	7	10000	3607	23418	16287	15,38	1,11
	Industry	8	10000	3447	23607	16306	14,73	1,13

Tabelle 4.111: Ergebnisse des Experiments Klass2IV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr1

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	1388	2452	2425	36,27	9,53
	Industry	6	2000	1397	2361	2413	36,92	8,55
	Industry	7	10000	7173	11949	11937	37,52	10,75
	Industry	8	10000	6947	12056	12071	36,54	9,42
Lewis-Feature-Vektoren	Industry	5	2000	873	2945	2991	22,73	4,51
	Industry	6	2000	920	2910	2928	23,96	5,38
	Industry	7	10000	4538	14472	14834	23,65	6,11
	Industry	8	10000	4605	14359	14614	24,12	5,96

Tabelle 4.112: Ergebnisse des Experiments Klass2IV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr2

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	1407	2426	2413	36,77	9,72
	Industry	6	2000	1433	2389	2391	37,48	10,16
	Industry	7	10000	7177	11879	11972	37,57	10,45
	Industry	8	10000	7029	12026	11998	36,92	10,30
Lewis-Feature-Vektoren	Industry	5	2000	894	2091	3031	25,88	6,26
	Industry	6	2000	822	2140	3104	23,87	5,54
	Industry	7	10000	4290	10626	15436	24,77	6,31
	Industry	8	10000	4185	10602	15420	24,34	6,11

Tabelle 4.113: Ergebnisse des Experiments Klass2IV für den Support-Vector-Machine-Klassifizierer für Trainingsmenge Tr3

Verfahren	Klassen-familie	Subset Test	Anzahl Testdaten	TP	FP	FN	micro F_1 in %	macro F_1 in %
TSF-Feature-Vektoren	Industry	5	2000	1395	2389	2426	36,69	10,45
	Industry	6	2000	1414	2331	2409	37,37	9,50
	Industry	7	10000	7235	11793	11926	37,89	10,56
	Industry	8	10000	7013	11991	12040	36,86	9,83
Lewis-Feature-Vektoren	Industry	5	2000	701	2515	3241	19,59	5,38
	Industry	6	2000	747	2473	3180	20,90	6,56
	Industry	7	10000	3753	12306	16017	20,95	6,17
	Industry	8	10000	3728	12346	15903	20,88	6,05

- Interpretation der Ergebnisse
  - Einführung
    Im Folgenden werden die vorgestellten Ergebnisse für die auf den Suffix Arrays basierenden Vektoren und für die Lewis-Feature-Vektoren einzeln pro Algorithmus miteinander verglichen. Abschließend wird ein allgemeiner Vergleich zwischen den zwei Verfahren bezogen auf das gesamte Experiment über alle Algorithmen und Trainingsteilmengen gezogen.

– Ergebnisvergleich Naive Bayes
Bildet man die Ergebnisse, die mit dem Naive-Bayes-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man Folgendes[1]:

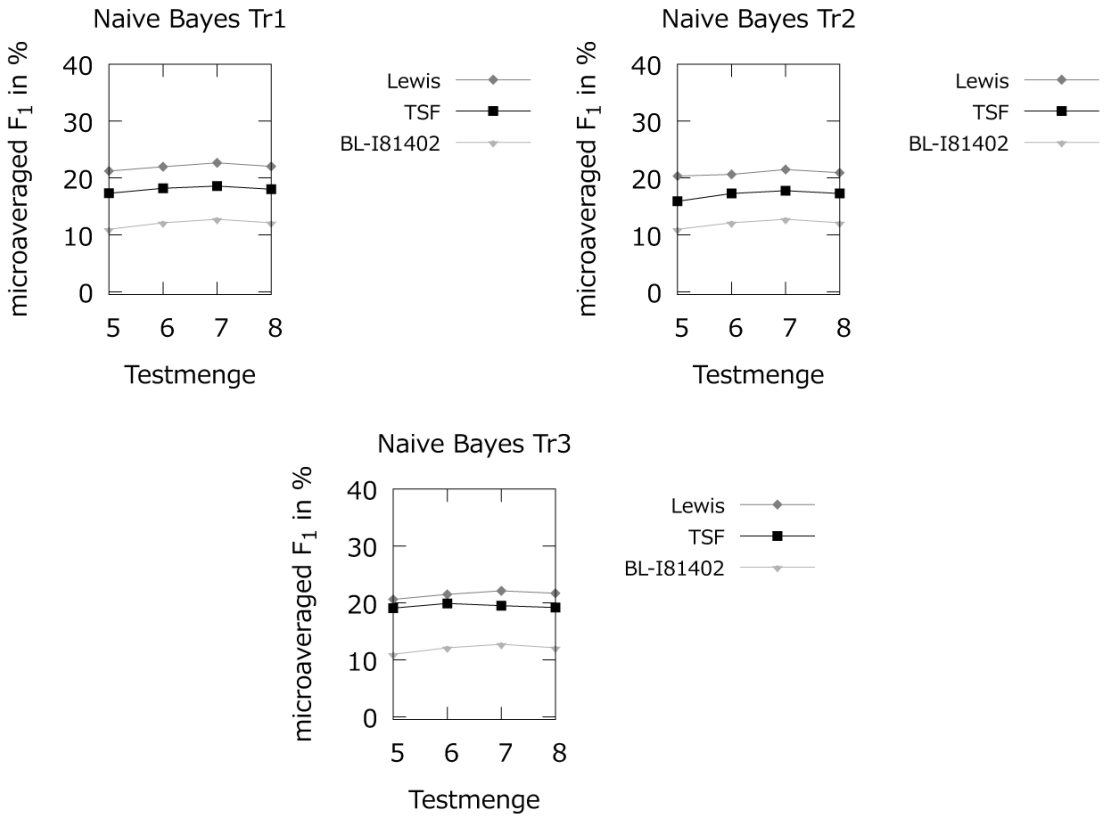


Abbildung 4.108: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den Naive-Bayes-Klassifizierer für das Experiment Klass2IV

Die abgebildeten Ergebnisse zeigen, dass einzelwortbasierte Features für den Naive-Bayes-Klassifizierer und die Klassenfamilie Industry eine bessere Qualität erreichen als die wortübergreifenden Features. Das gilt sowohl für den Fall, dass der Schwerpunkt auf Klassen mit vielen Dokumenten liegt, als auch für den Fall von Klassen mit wenigen Dokumenten. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen für den Microaveraged-Fall, so erhält man im

1 Die Abkürzungen in den Diagrammen bedeuten für dieses Experiment Folgendes: Baseline wird mit „BL“ abgekürzt und „I81402“ steht für den Klassennamen der Klasse mit den meisten Trainingsdokumenten.

Durchschnitt einen Abstand von 3,26 Prozentpunkten zwischen den mit den Einzelwort-Features und mit den TSF-Features erreichten Ergebnissen. Der minimale Abstand beträgt 1,55 Prozentpunkte und der maximale Abstand 4,41 Prozentpunkte. Im Fall der Macroaveraged-Auswertung ist der Abstand im Durchschnitt mit 1,20 Prozentpunkten noch geringer. Der minimale Abstand beträgt 0,06 Prozentpunkte und der maximale 1,99 Prozentpunkte. Beide Verfahren liegen also bei beiden Auswertungen des F_1-Maßes sehr dicht beieinander.

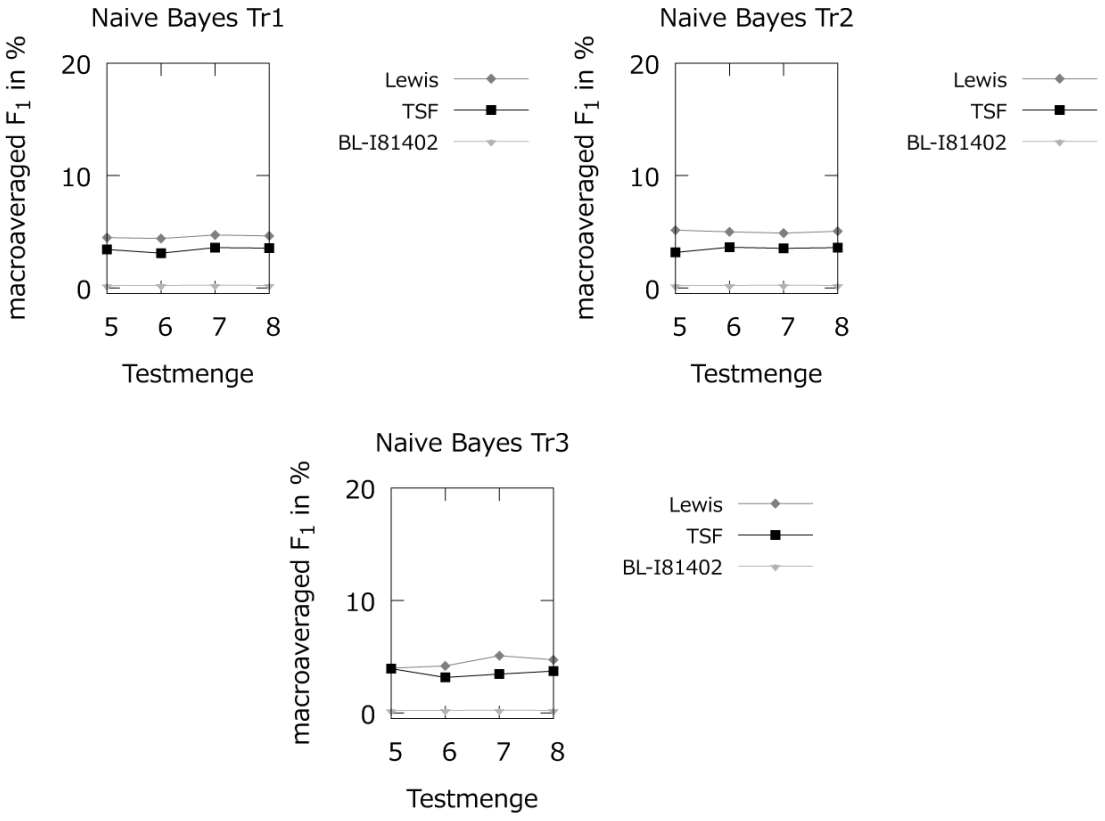


Abbildung 4.109: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den Naive-Bayes-Klassifizierer für das Experiment Klass2IV

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von ca. 20% im Fall der Microaveraged-Auswertung erreicht wird. Bei der Macroaveraged-Auswertung wird ein Wert von maximal 5% erreicht. Für beide Verfahren gilt, dass über alle Trai-

ningsteilmengen wesentlich bessere Ergebnisse erzielt werden als mit einer zufälligen Zuordnung der Testdokumente zu den zuordenbaren Klassen, d.h., es werden bessere Ergebnisse erreicht als mit der Baseline-Klassifizierung. Das gilt sowohl für den Fall, dass Klassen mit vielen Dokumenten mehr Bedeutung beigemessen wird, als auch für den Fall, dass Klassen mit wenigen Dokumenten mehr Gewicht beigemessen wird.

Ein möglicher Grund für das schlechtere Abschneiden der TSF-Feature-Vektoren gegenüber den einzelwortbasierten Vektoren beim Naive-Bayes-Algorithmus für das Experiment Klass2IV kann der gleiche sein, der bereits beim Experiment Klass1IV genannt wurde: Die Basis für die Features unterscheidet sich bei den beiden zu vergleichenden Verfahren.

– Ergebnisvergleich Decision Tree
Bildet man die Ergebnisse, die mit dem Decision-Tree-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man die Abbildungen 4.110 und 4.111 auf S. 444 und 445.

Die Ergebnisse des Decision-Tree-Klassifizierers liegen für beide Verfahren nah beieinander. So beträgt der kleinste Abstand zwischen beiden Verfahren 0,01 Prozentpunkte im Macroaveraged-Fall. Als maximaler Abstand für den Decision Tree ergibt sich ein Wert von 1,32 Prozentpunkten für diesen Fall. Für die Trainingsteilmengen Tr1 und Tr3 ist das einzelwortbasierte Verfahren besser, bei Tr2 erreichen beide Verfahren fast das gleiche Ergebnis. Im Durchschnitt bedeutet das, dass das einzelwortbasierte Verfahren um 0,65 Prozentpunkte besser abschneidet als das Verfahren mit TSF-Features, wenn der Schwerpunkt auf Klassen mit wenigen Dokumenten liegt.

Für die Auswertung mit Schwerpunkt auf Klassen mit vielen Dokumenten gilt das Gleiche. Im Durchschnitt erreicht das einzelwortbasierte Verfahren Ergebnisse, die um 2,1 Prozentpunkte besser sind als das wortübergreifende Verfahren. Der minimale Abstand beträgt hier 0,48 Prozentpunkte und der maximale 4,07 Abstand Prozentpunkte.

In beiden Fällen ist der gemessene Abstand zwar nicht so deutlich, aber er ist vorhanden. Es handelt sich also um den zweiten Algorithmus dieses Experiments, für den das einzelwortbasierte Verfahren bessere Ergebnisse erzielt als das Verfahren, welches auf TSF-Features basiert. Eine mögliche Erklärung dafür ist: *overfitting*.[1]

1 Siehe S. 58 dieser Arbeit.

Das Hauptaugenmerk in der vorliegenden Arbeit liegt auf dem Vergleich der erreichten Ergebnisse des wortübergreifenden Verfahrens und des einzelwortbasierten Verfahrens. Absolut gesehen zeigen die Klassifizierungsergebnisse, dass eine Qualität von ca. 15% im Microaveraged-Fall erreicht wird und eine Qualität von ca. 4% im Macroaveraged-Fall. Für beide Verfahren gilt, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung.

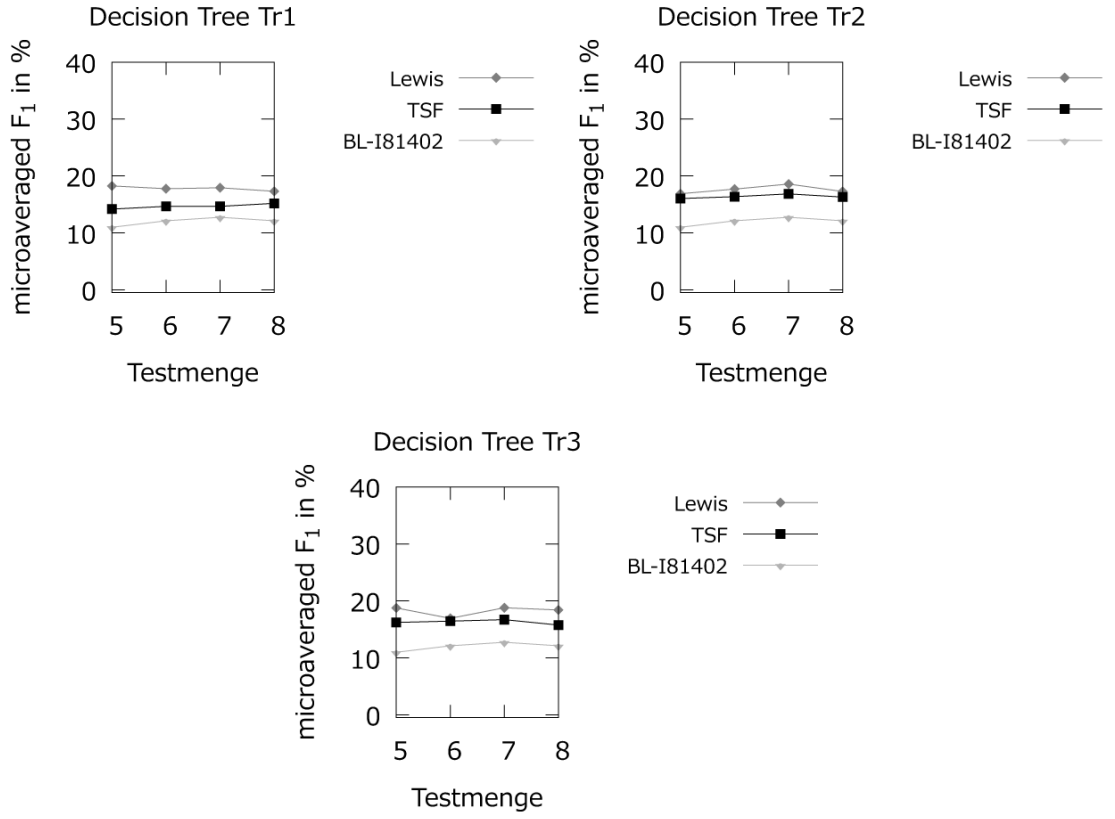


Abbildung 4.110: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den Decision-Tree-Klassifizierer für das Experiment Klass2IV

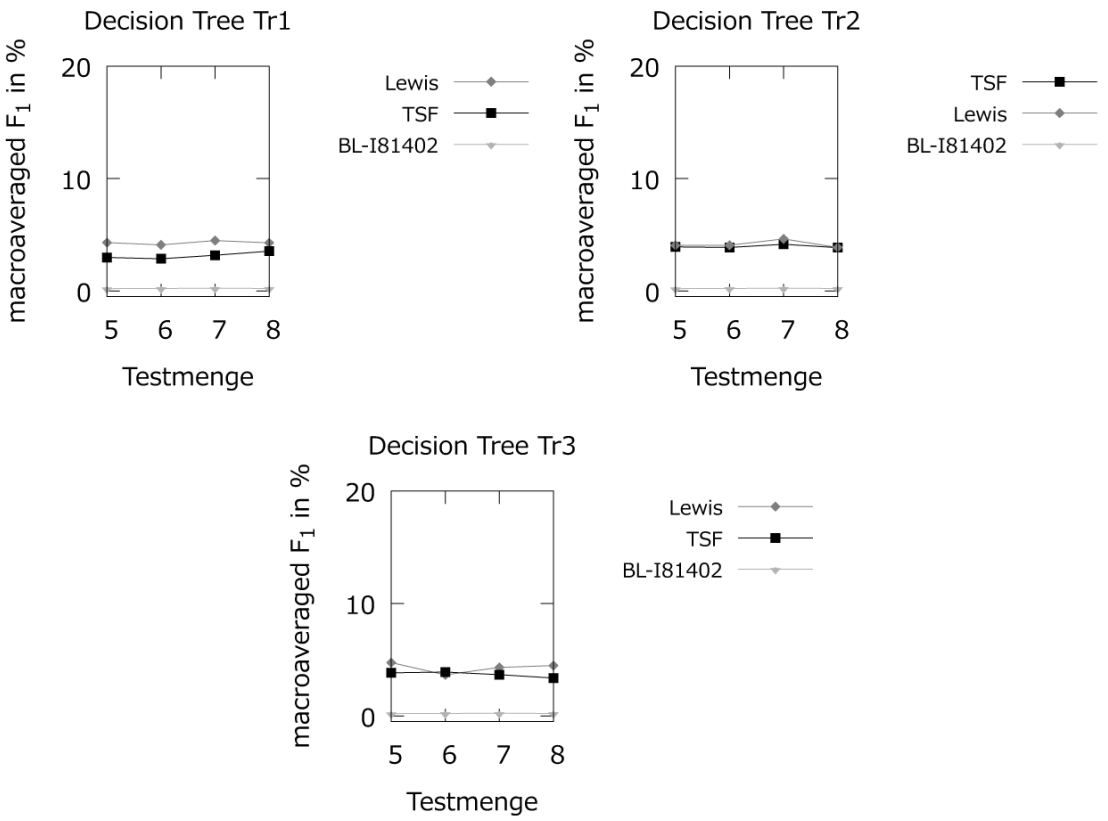


Abbildung 4.111: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den Decision-Tree-Klassifizierer für das Experiment Klass2IV

– Ergebnisvergleich k-Nearest-Neighbour
  Bildet man die Ergebnisse, die mit dem 10-Nearest-Neighbour-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man die Abbildungen 4.112 und 4.113 auf S. 446 und 447.

  Die abgebildeten Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Microaveraged-Fall im Durchschnitt einen Abstand von 17,40 Prozentpunkten zwischen den mit den TSF-Features und den Einzelwort-Features erreichten Ergebnissen. Der minimale Abstand beträgt 16,22 Prozentpunkte und der maximale Abstand 17,88 Prozentpunkte. Beim 10NN-Klassifizierer liegen die Ergebnisse beider Verfahren im Microaveraged-Fall weit auseinander.

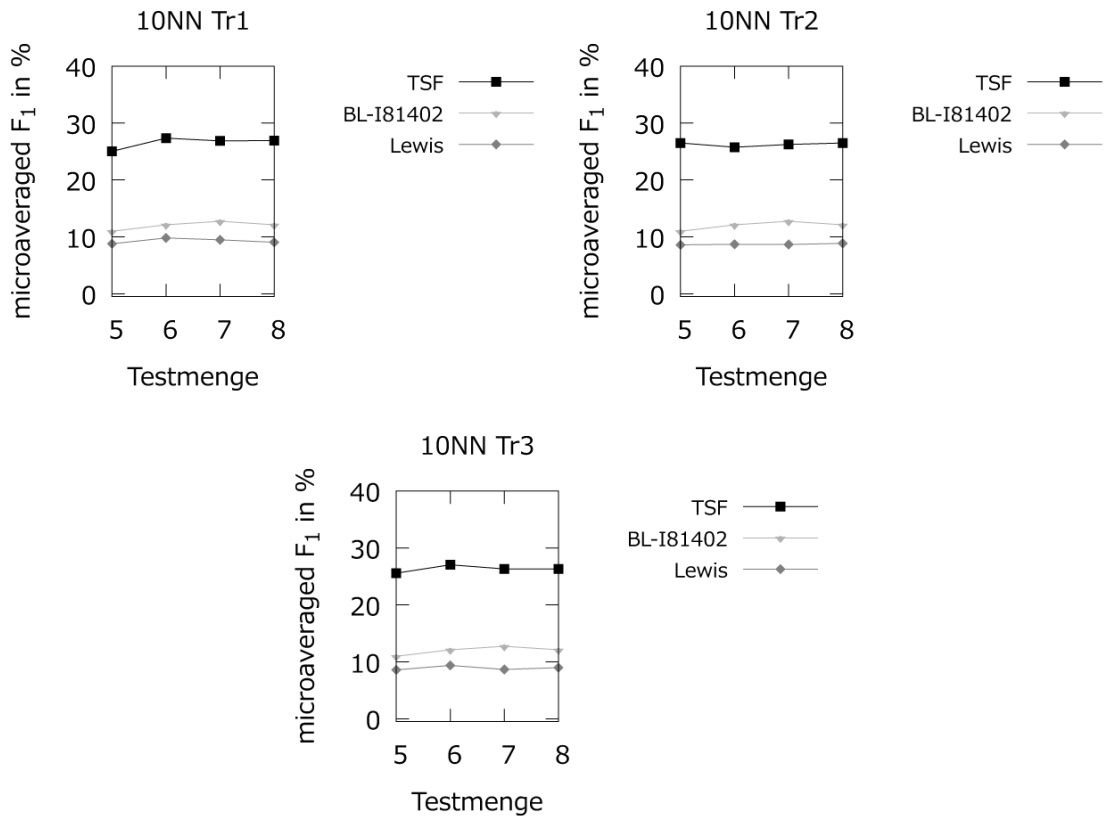


Abbildung 4.112: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den 10-Nearest-Neighbour-Klassifizierer für das Experiment Klass2IV

Auch im Macroaveraged-Fall besteht ein Abstand zwischen den beiden Verfahren, jedoch ist dieser nicht so groß wie im Microaveraged-Fall. Im Macroaveraged-Fall schneidet ebenfalls das wortübergreifende Verfahren besser ab als das einzelwortbasierte. Der minimale Abstand beträgt 2,44 Prozentpunkte und der maximale Abstand 4,07 Prozentpunkte. Im Durchschnitt bedeutet das einen Abstand von 3,29 Prozentpunkten.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von ca. 25% erreicht wird, wenn der Schwerpunkt auf Klassen mit vielen Dokumenten liegt. Liegt er dagegen auf Klassen mit wenigen Dokumenten, so wird eine Qualität von ca. 6% erreicht. Im erstgenannten Fall werden nur vom Verfahren mit den TSF-Feature-Vek-

toren über alle Trainingsteilmengen bessere Ergebnisse erzielt als mit der Baseline-Klassifizierung. Mit den einzelwortbasierten Feature-Vektoren klassifiziert der 10NN-Klassifizierer dagegen schlechter als eine zufällige Zuordnung der Dokumente zur Majority-Klasse. Liegt der Schwerpunkt bei der Auswertung des F_1-Maßes auf Klassen mit wenigen Dokumenten, so erreichen beide Verfahren bessere Ergebnisse als die Baseline-Klassifizierung.

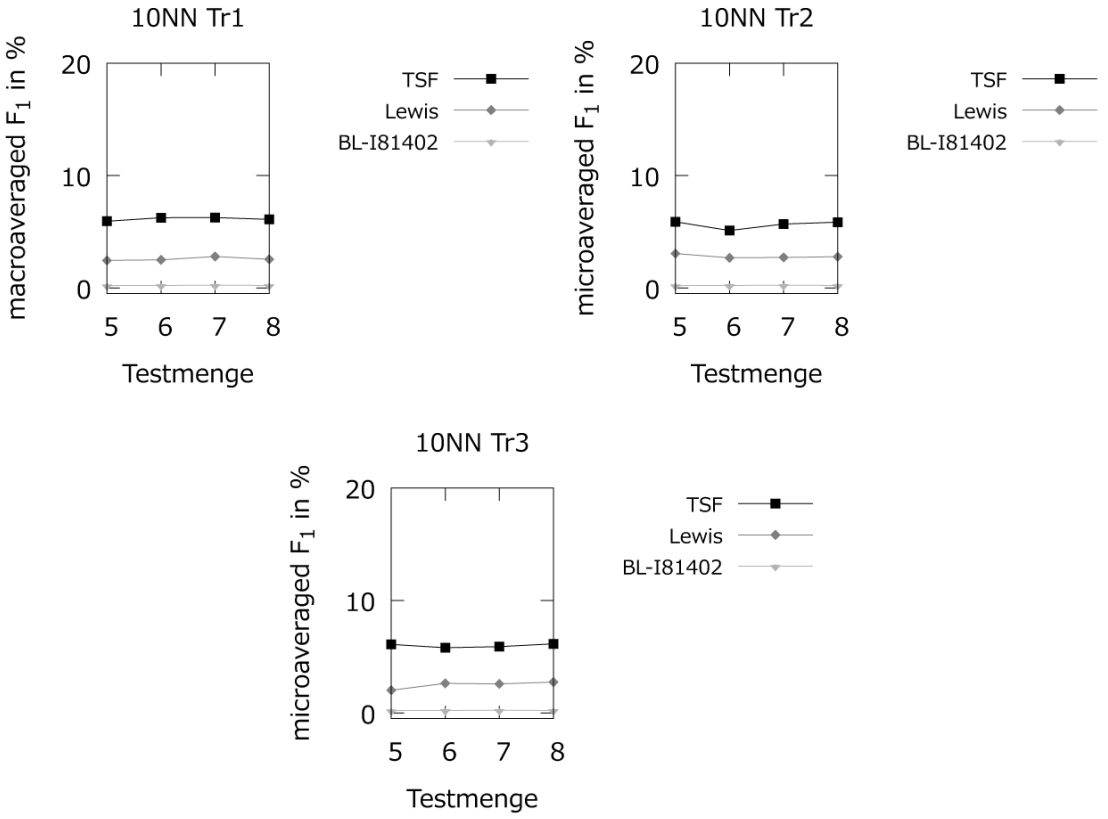


Abbildung 4.113: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den 10-Nearest-Neighbour-Klassifizierer für das Experiment Klass2IV

Bildet man die Ergebnisse, die mit dem 20-Nearest-Neighbour-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man die Abbildungen 4.114 und 4.115 auf S. 448 und 449.

Die abgebildeten Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trai-

ningsteilmengen für den Fall, dass der Schwerpunkt auf Klassen mit vielen Dokumenten liegt, so erhält man im Durchschnitt einen Abstand von 18,49 Prozentpunkten zwischen den mit den TSF-Features und den Einzelwort-Features erreichten Ergebnissen.

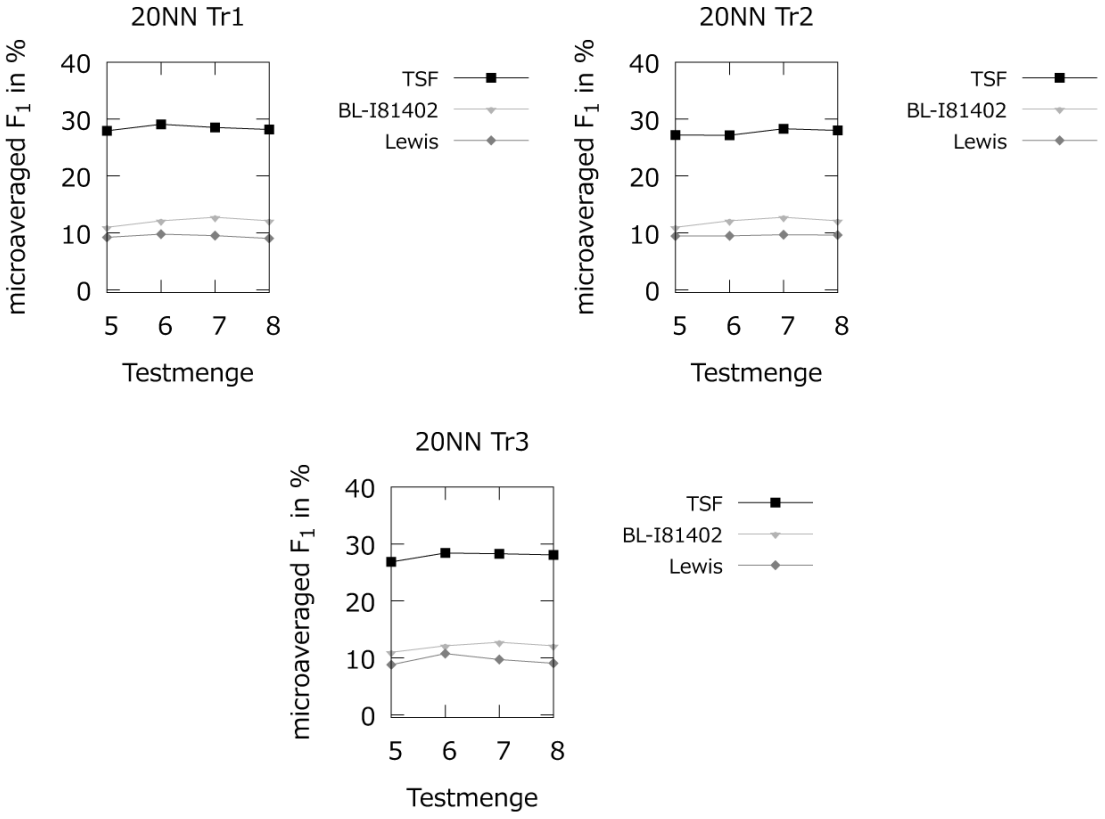


Abbildung 4.114: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den 20-Nearest-Neighbour-Klassifizierer für das Experiment Klass2IV

Das ist der bisher deutlichste gemessene Abstand in diesem Experiment. Der minimale Abstand beträgt 17,65 Prozentpunkte und der maximale Abstand 19,32 Prozentpunkte. Beim 20NN-Klassifizierer liegen die Ergebnisse beider Verfahren im Microaveraged-Fall weit auseinander.

Wird der Schwerpunkt auf die Betrachtung von Klassen mit wenigen Dokumenten gelegt, so ergibt sich ein ähnliches Bild. Auch hier schneiden die TSF-Feature-Vektoren besser ab als die einzelwortbasierten Feature-Vektoren. Im Schnitt ergibt sich ein Abstand von 3,11 Prozentpunkten mit einem

minimalen Abstand von 2,38 und einem maximalen Abstand von 3,65 Prozentpunkten.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch.

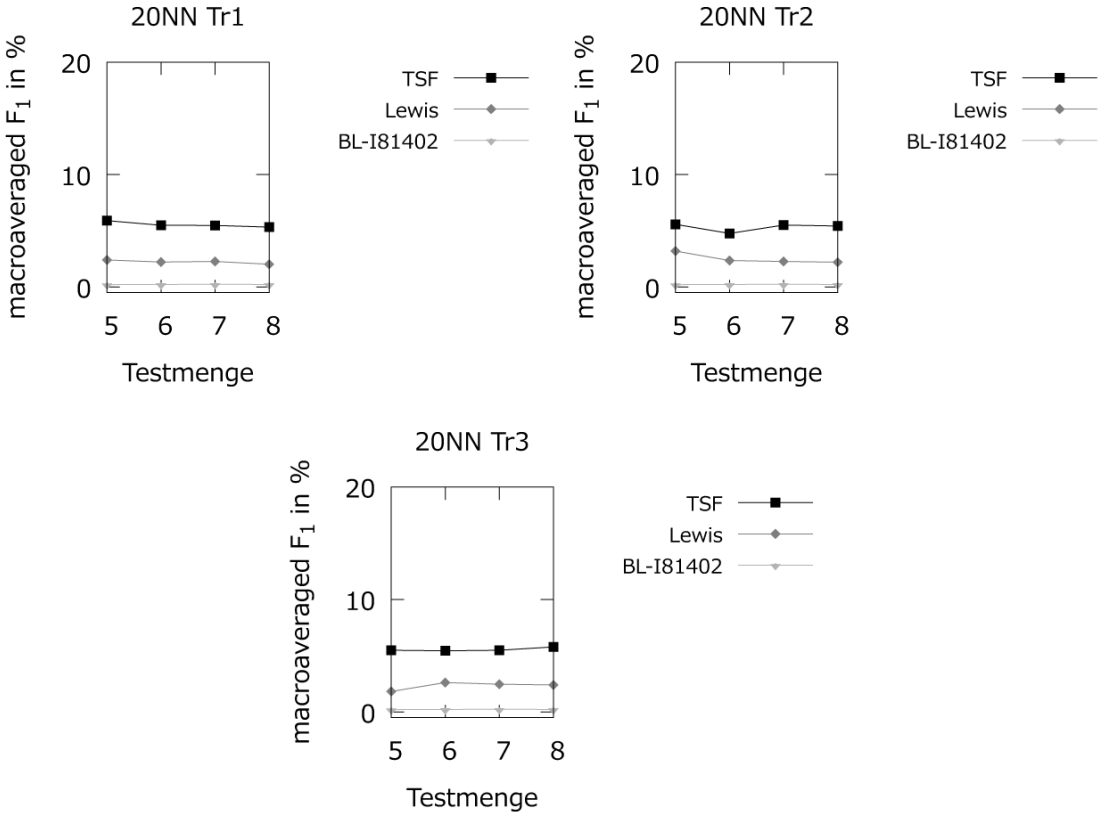


Abbildung 4.115: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den 20-Nearest-Neighbour-Klassifizierer für das Experiment Klass2IV

Die Klassifizierungsergebnisse zeigen, dass eine Qualität von ca. 30% für den Microaveraged-Fall erreicht wird. Für die TSF-Feature-Vektoren gilt hier, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung. Die einzelwortbasierten Feature-Vektoren erreichen dagegen schlechtere Ergebnisse als die Baseline-Klassifizierung. Im Macroaveraged-Fall werden Ergebnisse von ca. 5% erreicht. Hier erreichen beide Verfahren bessere Ergebnisse als die Baseline-Klassifizierung.

Bildet man die Ergebnisse, die mit dem 200-Nearest-Neighbour-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man die Abbildungen 4.116 und 4.117 auf S. 450 und 451.

Auch diese Ergebnisse stützen die der Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features.

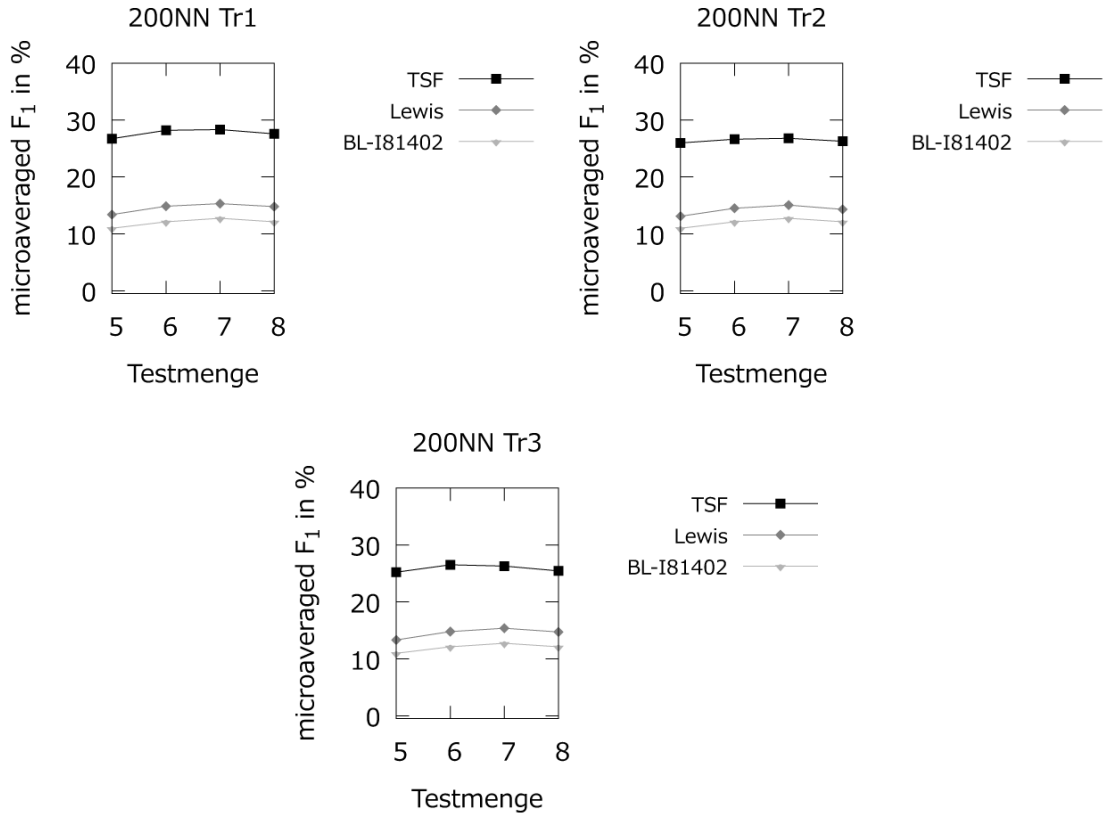


Abbildung 4.116: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den 200-Nearest-Neighbour-Klassifizierer für das Experiment Klass2IV

Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen für den Microaveraged-Fall, so erhält man im Durchschnitt einen Abstand von 12,19 Prozentpunkten zwischen den mit den TSF-Features und den Einzelwort-Features erreichten Ergebnissen. Der minimale Abstand beträgt 10,71 Prozentpunkte und der maximale Abstand 13,36 Prozentpunkte. Für den Macroaveraged-Fall liegen die Ergebnisse näher beieinander. Hier erreicht der 200NN-Klassifizierer für die

TSF-basierten Feature-Vektoren im Schnitt um 2,19 Prozentpunkte bessere Ergebnisse als für die einzelwortbasierten Feature-Vektoren. Der minimale Abstand beträgt dabei 1,74 Prozentpunkte und der maximale Abstand 2,42 Prozentpunkte.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal ca. 28% für den Fall, dass der Schwerpunkt auf Klassen mit vielen Dokumenten liegt, erreicht wird. Für beide Verfahren gilt, dass über alle Trainingsteilmengen bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung. Letzteres gilt auch im Macroaveraged-Fall. Hier liegen die Ergebnisse jedoch maximal bei ca. 3,6%.

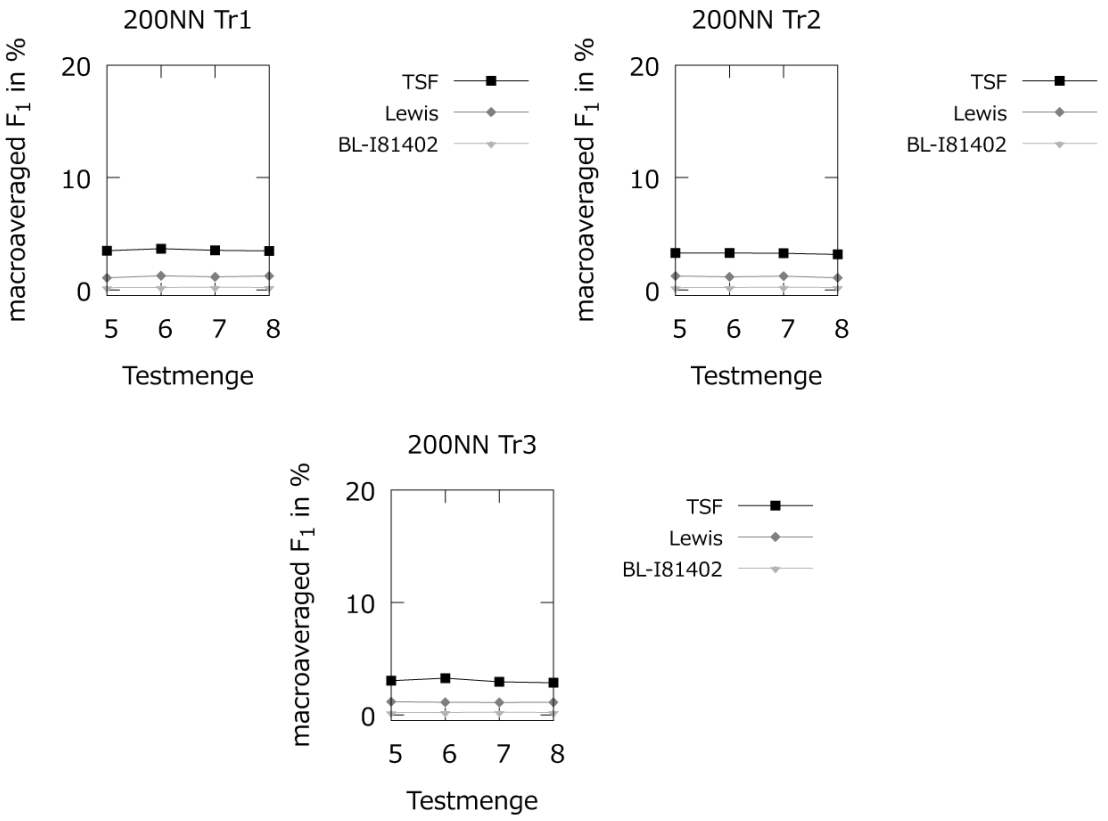


Abbildung 4.117: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den 200-Nearest-Neighbour-Klassifizierer für das Experiment Klass2IV

Insgesamt ergibt sich für die k-Nearest-Neighbour-Klassifizierer folgendes Bild:

* Die TSF-Feature-Vektoren erreichen in allen Fällen bessere Ergebnisse als die einzelwortbasierten Feature-Vektoren.
* Die Anzahl der Nachbarn scheint für die TSF-Feature-Vektoren in diesem Experiment keine Rolle zu spielen, da über alle Anzahlen von Nachbarn ähnliche Ergebnisse erreicht werden. Lediglich im Macroaveraged-Fall für den 200NN-Klassifizierer sinkt die Qualität der erzeugten Klassifizierung. Bei den einzelwortbasierten Feature-Vektoren scheint bei diesem Experiment für den kNN-Algorithmus eine höhere Anzahl von Nachbarn vorteilhafter zu sein, da dann bessere Ergebnisse erreicht werden.

– Ergebnisvergleich Support Vector Machine
Bildet man die Ergebnisse, die mit dem Support-Vector-Machine-Klassifizierer basierend auf den Trainingsteilmengen erzeugt wurden, in Diagrammen ab, so erhält man die Abbildungen 4.118 und 4.119 auf S. 453 und 454.

Die abgebildeten Ergebnisse stützen die dieser Arbeit zu Grunde liegende Vermutung, dass wortübergreifende Features bessere Klassifizierungsergebnisse liefern als einzelwortbasierte Features. Berechnet man die Abweichung zwischen den Ergebnissen der jeweiligen Testteilmengen über alle drei Trainingsteilmengen, so erhält man im Durchschnitt für die Microaveraged-Auswertung des F_1-Maßes, einen Abstand von 14,1 Prozentpunkten zwischen den mit den TSF-Features und mit den Einzelwort-Features erreichten Ergebnissen. Der minimale Abstand beträgt 10,89 Prozentpunkte und der maximale Abstand 17,1 Prozentpunkte. Auch im Macroaveraged-Fall erreicht der Klassifizierer mit wortübergreifenden Features bessere Ergebnisse als mit einzelwortbasierten Features. Hier beträgt der minimale Abstand 2,94 Prozentpunkte und der maximale Abstand 5,07 Prozentpuntke. Das bedeutet einen durchschnittlichen Abstand von 4,08 Prozentpunkten.

Zwar werden die Klassifizierungsergebnisse hauptsächlich für den Vergleich zwischen den beiden Verfahren betrachtet und nicht absolut, einige Anmerkungen zum absoluten Ergebnis folgen hier dennoch. Die Klassifizierungsergebnisse zeigen, dass eine Qualität von maximal ca. 35% im Fall der Microaveraged-Auswertung des F_1-Maßes erreicht wird. Für beide Verfahren gilt, dass über alle Trainingsteilmengen wesentlich bessere Ergebnisse erzielt werden als mit der Baseline-Klassifizierung. Im Fall der Macroaveraged-Auswertung des F_1-Maßes wird eine Qualität der Klassifizierung von maximal

ca. 10% erreicht. Auch hier übertrifft der Klassifizierer die Ergebnisse der randomisierten Zuordnung der Dokumente für beide Verfahren.

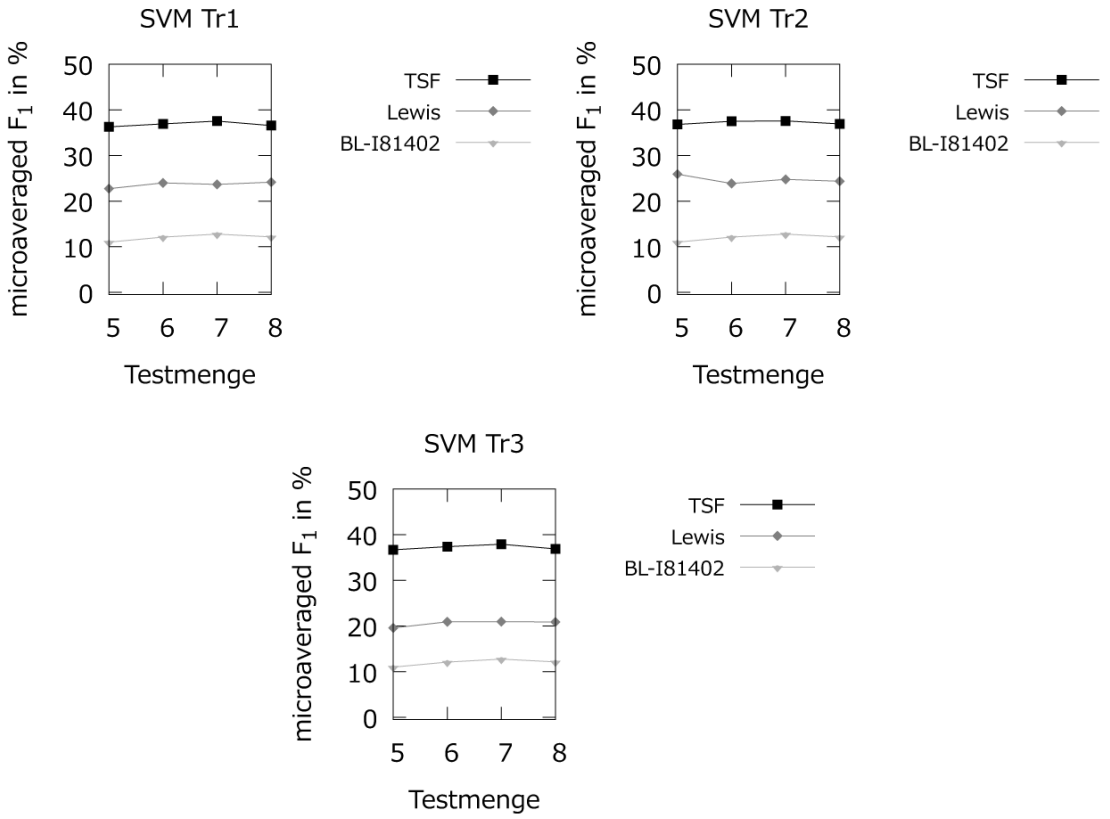


Abbildung 4.118: Microaveraged-Ergebnis der Evaluation der Klassifizierung durch den Support-Vector-Machine-Klassifizierer für das Experiment Klass2IV

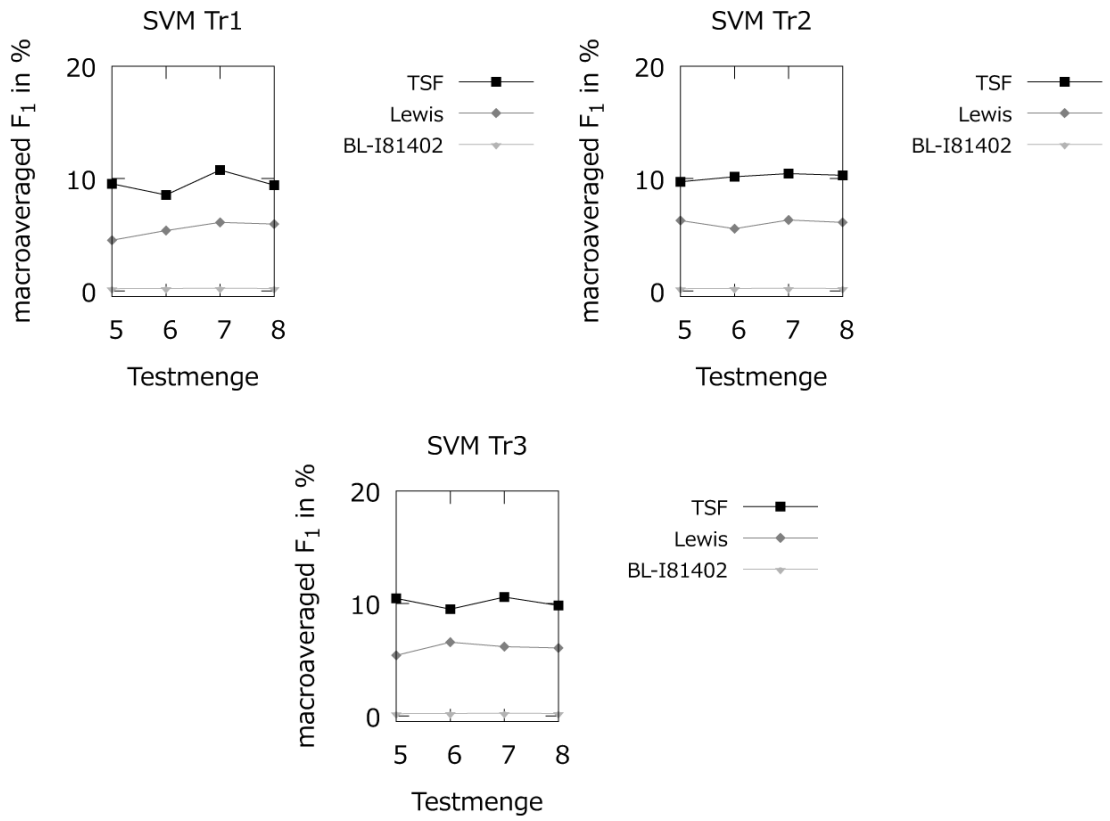


Abbildung 4.119: Macroaveraged-Ergebnis der Evaluation der Klassifizierung durch den Support-Vector-Machine-Klassifizierer für das Experiment Klass2IV

– Ergebnisvergleich über alle Algorithmen für das Experiment
Vergleicht man die Ergebnisse des einzelwortbasierten Verfahrens und des Verfahrens mit TSF-Features über alle Algorithmen miteinander, so stellt man Folgendes fest: Das auf TSF-Features basierende Verfahren schneidet beim Klassifizieren von natürlichsprachlichen Dokumenten für fast alle verwendeten Algorithmen besser ab als das einzelwortbasierte Verfahren. Die beiden Algorithmen, für die das nicht zutrifft, sind der Naive-Bayes-Algorithmus und der Decision-Tree-Algorithmus. Eine mögliche Ursache für das schlechtere Abschneiden der TSF-Feature-Vektoren wurde bereits in den entsprechenden Unterkapiteln genannt.

Als Tabelle zusammengefasst ergibt sich für das sechste Experiment folgendes Bild[1]:

Tabelle 4.114: Ergebnisse des Experiments Klass2IV

Verfahren	Algorithmus							
	NB (micro)	NB (macro)	DT (micro)	DT (macro)	kNN (micro)	kNN (macro)	SVM (micro)	SVM (macro)
TSF-Feature-Vektoren					✓	✓	✓	✓
Lewis-Feature-Vektoren	✓	✓	✓	✓				

Damit erreichen die TSF-Feature-Vektoren in 8 von 12 Fällen ein besseres Klassifizierungergebnis als die einzelwortbasierten Feature-Vektoren.[2]

Somit stützt dieses sechste Experiment die Vermutung, dass wortübergreifende Features qualitativ bessere Ergebnisse beim Klassifizieren von natürlichsprachlichen Dokumenten liefern als Verfahren, die natürlichsprachliche Dokumente in Einzelworte aufteilen und diese als Features für das Klassifizieren benutzen.

1 Ein Häkchen zeigt dabei das Verfahren an, das für den betreffenden Algorithmus die höchste Qualität über alle Trainings- und Testteilmengen erreicht hat.
2 Hierbei werden die unterschiedlichen Anzahlen an Nachbarn beim kNN einzeln gezählt.

4.3.2.2.4 Vergleich der Ergebnisse der Experimente 4 bis 6

Vergleicht man die erreichten Ergebnisse für die TSF-Feature-Vektoren und die einzelwortbasierten Vektoren über alle Klassenfamilien, Algorithmen und Auswertungsarten des F_1-Maßes miteinander, so erhält man Folgendes:

- Trotz der anderen Evaluation, die es zulässt, unterschiedliche Schwerpunkte bei der Auswertung der Klassifizierungsergebnisse zu setzen, ergibt sich auch für die Experimente 4 bis 6 das gleiche Bild wie für die Experimente 1 bis 3.
- In 30 von 36 Fällen erreichen die TSF-Feature-Vektoren bessere Ergebnisse beim Klassifizieren von natürlichsprachlichen Dokumenten als einzelwortbasierte Feature-Vektoren. In Prozent umgerechnet ergibt sich ein Wert von gerundet 83%. Damit kann man, basierend auf diesen Experimenten, sagen, dass die TSF-Feature-Vektoren besser für das Klassifizieren von natürlichsprachlichen Dokumenten geeignet sind als einzelwortbasierte Vektoren.
- Insbesondere der kNN-Algorithmus, aber auch der SVM-Algorithmus, scheinen besonders gut mit TSF-Feature-Vektoren klassifizieren zu können, da mit diesen Algorithmen durchweg die besten Ergebnisse erzielt werden. Der Naive-Bayes-Algorithmus erreicht in den genannten Fällen vermutlich aufgrund der Nicht-Beachtung aller Trainingsdokumente bei den TSF-Features im Gegensatz zu den einzelwortbasierten Features schlechtere Ergebnisse und der Decision-Tree-Algorithmus vermutlich aufgrund der Überanpassung an die Trainingsteilmengen.

4.3.2.3 Vergleich der Ergebnisse des Klassifizierens über alle Experimente

Fasst man die Ergebnisse aller Experimente noch einmal überblicksartig zusammen so ergibt sich Folgendes:

- In 83% der durchgeführten Experimente erreichen die wortübergreifenden Feature-Vektoren bessere Klassifizierungsergebnisse als die einzelwortbasierten Feature-Vektoren. Damit kann man, basierend auf diesen Experimenten, sagen, dass die TSF-Feature-Vektoren besser für das Klassifizieren von natürlichsprachlichen Dokumenten geeignet sind als einzelwortbasierte Feature-Vektoren.

- Die 17% der Fälle, für die das nicht gilt, haben zwei Gemeinsamkeiten:
  - Sie treten nur bei zwei Algorithmen auf: dem Decision-Tree-Algorithmus und dem Naive-Bayes-Algorithmus.
  - Der durchschnittliche Abstand zwischen den Ergebnissen in diesen Fällen liegt bei unter 5%. D.h., in allen Fällen, in denen die Klassifizierer mit den einzelwortbasierten Feature-Vektoren besser abschneiden als diejenigen mit den TSF-Feature-Vektoren, ist das Ergebnis mit den einzelwortbasierten Feature-Vektoren maximal um 5% besser. Das trifft in den anderen 83% nicht zu. Die einzige Ausnahme ist die Macroaveraged-Auswertung des F_1-Maßes.
- In den Fällen, in denen die Klassifizierer mit den TSF-Feature-Vektoren besser sind als die mit den einzelwortbasierten Feature-Vektoren, unterscheiden sich die Ergebnisse sehr deutlich voneinander. Ausnahmen sind einige Macroaveraged-Ergebnisse insbesondere bei der Klassenfamilie Industry und die Ergebnisse des Support-Vector-Machine-Klassifizierers bei der Topic-Klassenfamilie mit einer Evaluation des Anteils der korrekten Dokumente an der Gesamtmenge der Dokumente.

Zusammenfassend lässt sich sagen, dass die Klassifizierer mit einzelwortbasierten Feature-Vektoren in keinem der durchgeführten Experimente einen ähnlich großen Abstand zu den Ergebnissen der Klassifizierer mit TSF-Feature-Vektoren erreichen wie umgekehrt. Man kann also sagen, dass die Klassifizierer mit wortübergreifenden Feature-Vektoren in nahezu allen durchgeführten Experimenten annähernd gleiche oder bessere Ergebnisse erreichen wie die gleichen Klassifizierer mit einzelwortbasierten Feature-Vektoren.

5 Experimente mit dem Clustern von natürlichsprachlichen Dokumenten

5.1 Implementierung des Clusterns mit Text-Suffix-Fragment-Features

5.1.1 Einleitung der Implementierung des Clusterns mit Text-Suffix-Fragment-Features

Ein Ziel der vorliegenden Arbeit ist es zu überprüfen, ob erzeugte Clusterings von natürlichsprachlichen Dokumenten basierend auf TSF-Features bessere Ergebnisse liefern als erzeugte Clusterings der gleichen Dokumente basierend auf Einzelworten als Features. Nachdem im vorangegangenen Kapitel beschrieben wurde, wie mit TSF-Features klassifiziert wird, folgt in diesem Kapitel die Beschreibung des Ablaufs, wie mit diesen Features Dokumente geclustert werden.
Dabei wird in jedem Unterkapitel ein Schritt des Ablaufs innerhalb der Implementierung[1] für das Verfahren mit TSF-Features erläutert. Die Beschreibung des Ablaufs für das einzelwortbasierte Verfahren folgt an späterer Stelle.

5.1.2 Daten vorbereiten für das Clustern mit Text-Suffix-Fragment-Features

5.1.2.1 Teilmengen erstellen für das Clustern mit Text-Suffix-Fragment-Features

Beim Clustern wird ebenfalls der Reuters-RCV1-v2-Datensatz verwendet. Eine Beschreibung, wie dieser erstellt und aufbereitet wird, befindet sich in Kapitel 4.1.3 bis 4.1.6 der vorliegenden Arbeit. Die Aufteilung der dort vorhandenen 23.149 Trainingsdaten[2] in Teilmengen erfolgt auch beim Clustern, um die Durchführungsgeschwindigkeit der verschiedenen Experimente zu erhöhen. Die genauen Parameter, also wie viele Teilmengen erstellt werden, wie viele Dokumente diese umfassen und welche

1 Eine Erläuterung zum Software-Prototyp befindet sich in Anhang F auf S. 593.
2 Beim Clustern wird in der vorliegenden Arbeit von Daten gesprochen. Es wird keine Aufteilung in Trainings- und Testdatensatz benötigt, da kein Training stattfindet, sondern das Clustering aus den Daten des RCV1-v2-Datensatzes direkt erzeugt und evaluiert wird. Die Grundlage dieser Erzeugung liefern die Trainingsdaten des RCV1-v2.

Anzahl an Klassenzugehörigkeiten die ausgewählten Dokumente haben, können den Beschreibungen der jeweiligen Experimente entnommen werden, siehe Kapitel 5.3. Die Teilmengen sind durch ein Verzeichnis symbolisiert, in dem sich die ausgewählten, konvertierten Reuters-Daten, die zur Teilmenge gehören, befinden. Sie sind als serialisierte Java-Objekte der Java-Klasse `FileToCategorize` vorhanden.

5.1.2.2 Suffix Arrays erstellen für das Clustern mit Text-Suffix-Fragment-Features

Für jede dieser Teilmengen der Daten, die auf die zuvor beschriebene Weise erstellt worden sind, wird für jedes enthaltene und durch das `FileToCategorize`-Objekt repräsentierte Dokument ein Suffix Array erzeugt. Die Erzeugung erfolgt, wie in Kapitel 3.1.4.4 beschrieben. Diese erzeugten Suffix Arrays werden als serialisierte Java-Objekte der Java-Klasse `SuffixArrayToSave` in Bezug auf die Teilmenge, zu der sie gehören, gespeichert.

5.1.2.3 Text-Suffix-Fragment-Features ermitteln für das Clustern mit Text-Suffix-Fragment-Features

Beim Clustern mit TSF-Features werden zwei Ansätze beim Extrahieren der Features verwendet. TSF-Features müssen zwei Kriterien genügen: 1. Sie müssen mindestens doppelt vorkommen und 2. sie müssen mindestens drei Zeichen lang sein. Beim Klassifizieren werden TSF-Features ermittelt, die mindestens doppelt in Dokumenten der Trainingsdaten mit der gleichen Klasse vorkommen. Beim Clustern werden die Klassen der Dokumente nicht verwendet. Sie sind also zum Zeitpunkt des Clusterns nicht bekannt. Daher muss hier eine andere Vorgehensweise gewählt werden. Die zwei hier verwendeten Ansätze sind:

1. Ermitteln der TSF-Features aller Dokumente in den zu clusternden Daten.
   Dieser Ansatz bedeutet, dass ein TSF-Feature über *alle* Dokumente der zu clusternden Teilmenge mindestens doppelt vorkommen muss. Also taucht es doppelt in mindestens einem Dokument auf, oder aber in mindestens zwei verschiedenen Dokumenten.

2. Ermitteln der TSF-Features jedes einzelnen Dokuments in den zu clusternden Daten.
   Dieser Ansatz bedeutet, dass ein TSF-Feature in *einem* Dokument doppelt vorkommen muss.

Angestoßen wird das Ermitteln der TSF-Features durch die Operation `extractFeaturesForClustering`, zu sehen in Abbildung 5.1. Sie ruft abhängig vom gewählten Ansatz entweder die Operation `extractFeaturesOfAllDoks` auf, um die Features über alle Dokumente zu suchen, oder die Operation `extractFeaturesOfSingleDoks`, um in jedem einzelnen Dokument die Features zu suchen.

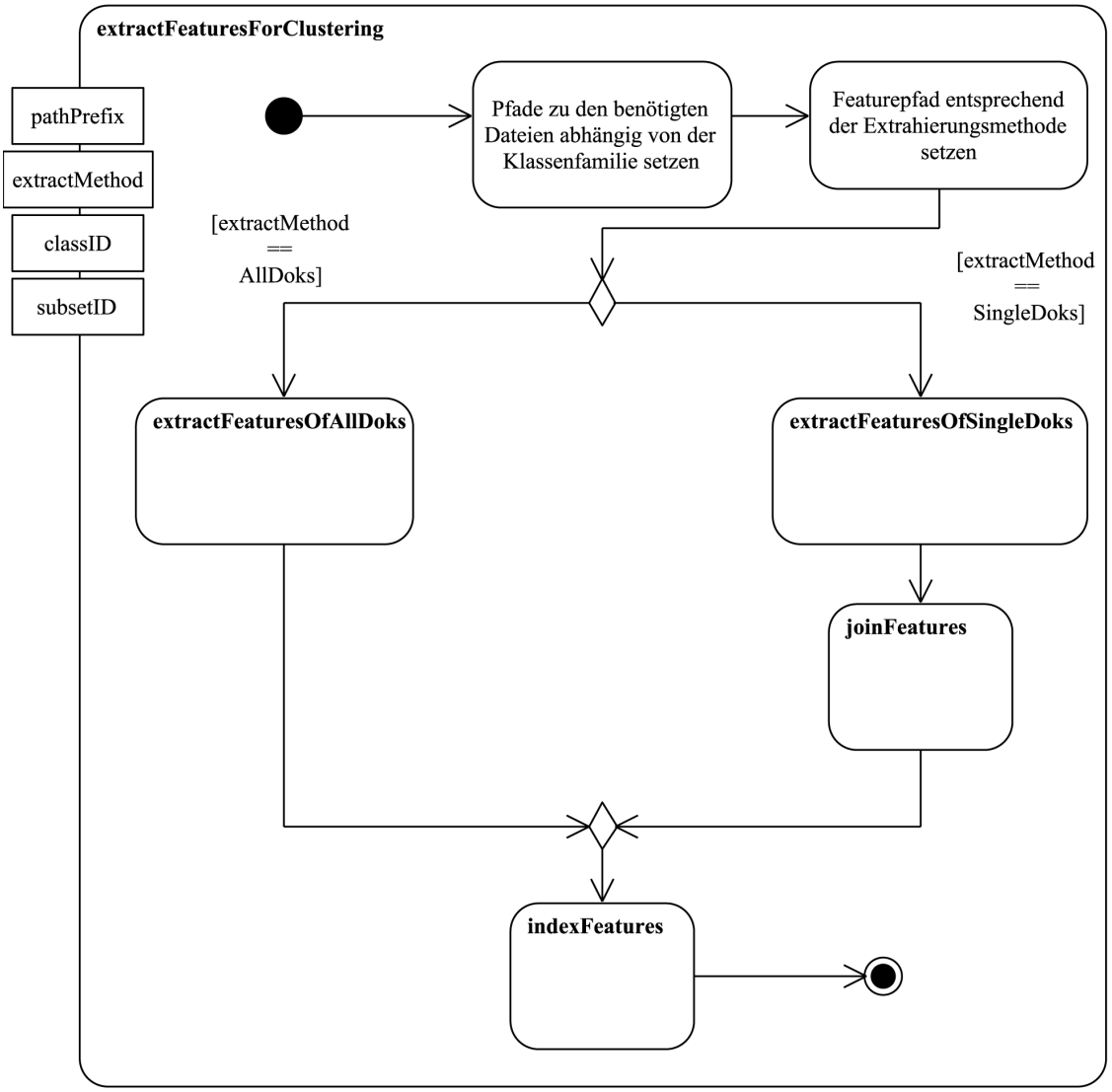


Abbildung 5.1: Ablauf der Operation `extractFeaturesForClustering`

Wird der erste Ansatz verwendet, so erfolgt das Ermitteln der Features ähnlich wie beim Klassifizieren. Alle einzelnen Suffix Arrays der Teilmenge werden in einem generalisierten Suffix Array zusammengefasst. Das entspricht der Vorgehensweise beim Klassifizieren, wenn alle einzelnen Suffix Arrays einer Klasse zusammengefasst werden. Anschließend wird die Operation `extractFeatures` des generalisier-

ten Suffix Arrays aufgerufen, um die TSF-Features zu ermitteln. Die Operation `extractFeatures` ist zu sehen in Abbildung 3.36 auf S. 224 dieser Arbeit. Sie wird auch im entsprechenden Kapitel beschrieben. Daher wird für eine Beschreibung auf das Kapitel 3.2.3.1 dieser Arbeit verwiesen. Zum Abschluss der Operation `extractFeaturesOfAllDoks` werden das generalisierte Suffix Array und die ermittelten Features gespeichert. Zu sehen ist die Operation `extractFeaturesOfAllDoks` in Abbildung 5.2.

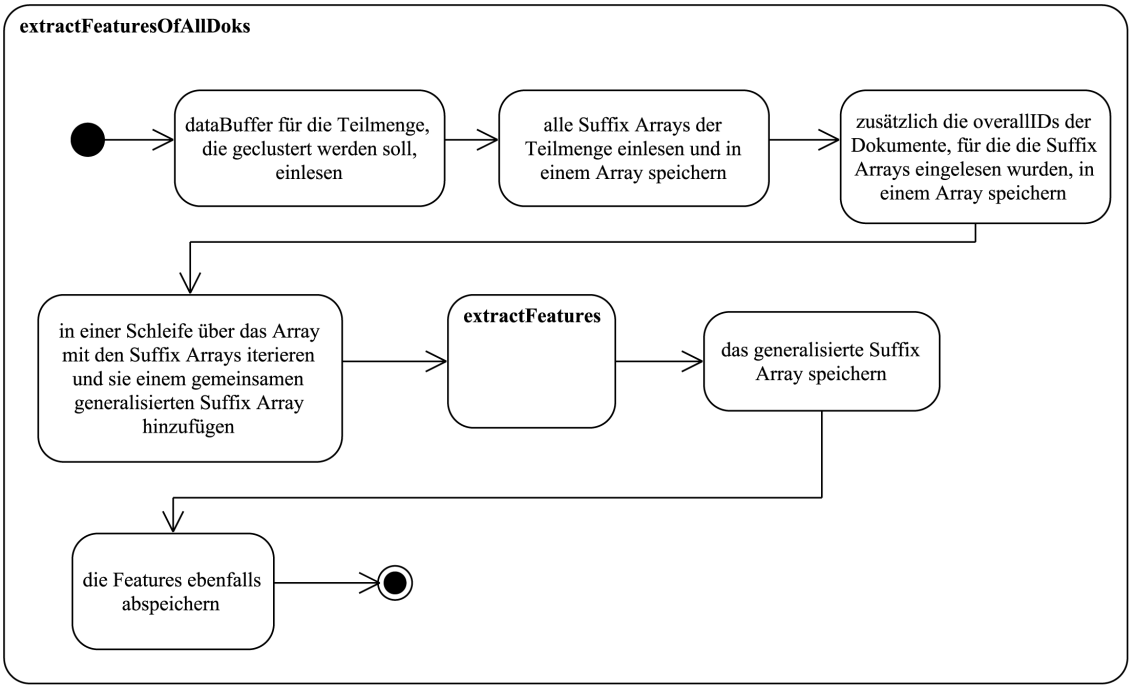


Abbildung 5.2: Ablauf der Operation `extractFeaturesOfAllDoks`

Wird der zweite Ansatz gewählt, so ist der Ablauf etwas anders. Auch in diesem Fall wird ein generalisiertes Suffix Array verwendet, um die TSF-Features zu ermitteln. Diesmal werden jedoch nicht alle einzelnen Suffix Arrays zu einem generalisierten Suffix Array zusammengeführt, sondern für jedes einzelne Suffix Array der Teilmenge ein eigenes generalisiertes Suffix Array erstellt. Es handelt sich also lediglich um eine Umwandlung der Datenstruktur, um die bereits definierten Operationen wiederverwenden zu können, d.h., nach der Umwandlung eines einzelnen Suffix Arrays in ein generalisiertes Suffix Array werden dort die TSF-Features ermittelt und abgespeichert. Danach wird mit dem nächsten Suffix Array fortgefahren. Zu sehen ist der Ablauf in Abbildung 5.3 auf S. 462.

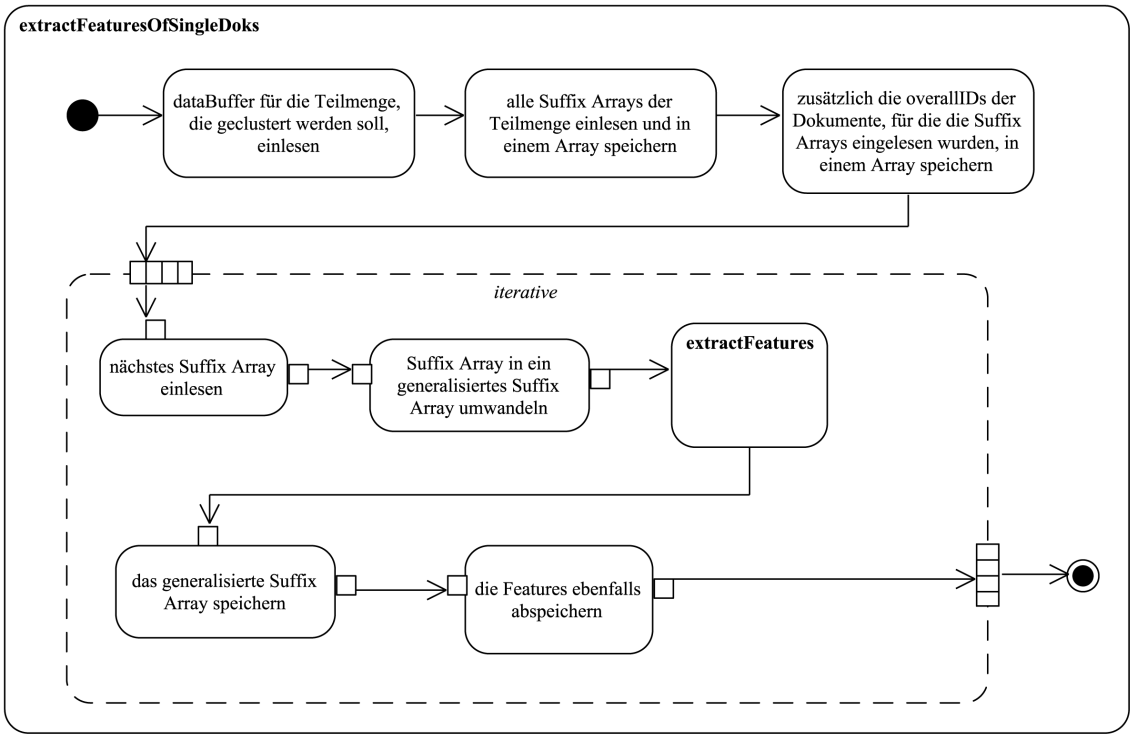


Abbildung 5.3: Ablauf der Operation `extractFeaturesOfSingleDoks`

Um bei der späteren Erstellung der Vektoren eindeutige Features zur Verfügung zu haben, muss beim zweiten Ansatz *eine* Datei mit allen TSF-Features aller Dokumente erstellt werden. Das vermeidet die mehrfache Betrachtung des gleichen Features. Aus diesem Grund werden in der Operation `joinFeatures`, zu sehen in Abbildung 5.4 auf S. 463, alle Feature-Objekte der einzelnen Dokumente der Teilmenge eingelesen und in einem Java-Objekt der Java-Klasse `Hashtable` vereinigt. Durch die Festlegung des Features als Schlüssel der Hashtabelle kann jedes Feature nur ein einziges Mal dort vorhanden sein. Der Wert, also das Objekt der Klasse `PhraseData`, muss dagegen aktualisiert werden. Das geschieht, indem die dort vorhandenen Daten zusammengeführt werden. Abschließend wird das Objekt mit allen Features in einer Datei abgelegt.

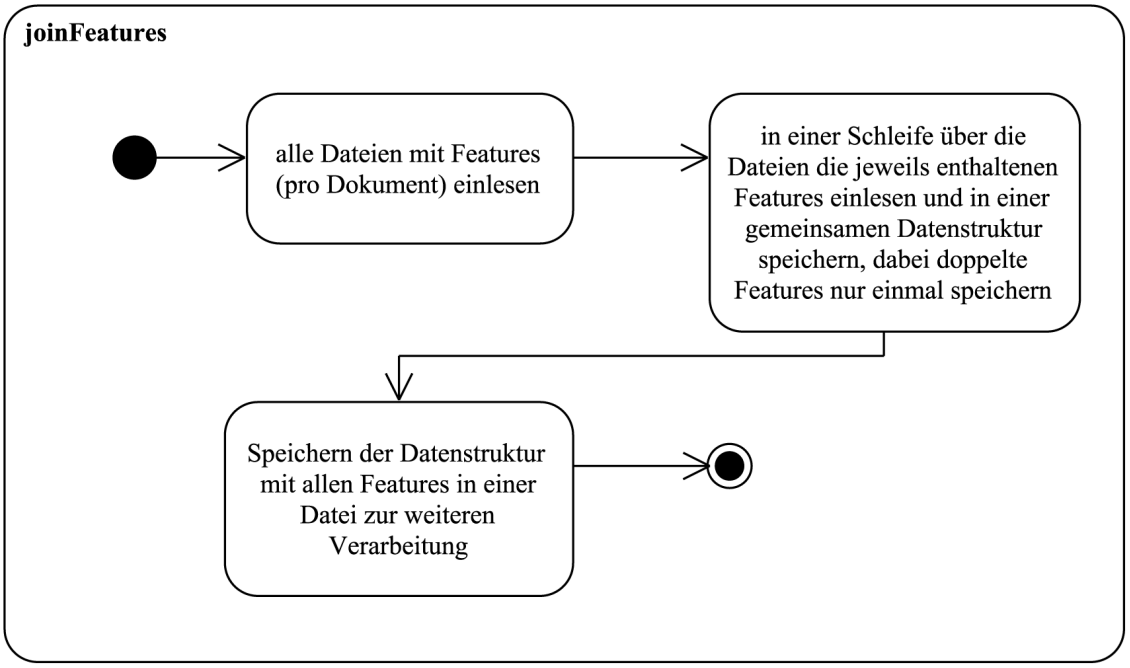


Abbildung 5.4: Ablauf der Operation `joinFeatures`

5.1.2.4 Indizieren der Text-Suffix-Fragment-Features für das Clustern mit Text-Suffix-Fragment-Features

Zur Erzeugung der Vektoren müssen die Dokumente daraufhin überprüft werden, ob die ermittelten Features in ihnen enthalten sind. Um diesen Vorgang zu erleichtern, werden die ermittelten Features bei beiden Ansätzen innerhalb der Operation `indexFeatures`, die auf die in Kapitel 4.1.7.7 der vorliegenden Arbeit erläuterte Operation `generateIndexes` zurückgreift, indiziert. Zu sehen ist die letztgenannte Operation in Abbildung 4.25 auf S. 255.

5.1.2.5 Erzeugen der Text-Suffix-Fragment-Feature-Vektoren für die Daten

Unter Zuhilfenahme des Index auf den Features und der Suffix Arrays der einzelnen Dokumente ist es nun möglich, für jedes Dokument einen Vektor zu erstellen. Jeder Dokumentenvektor zeigt an, welche Features über alle Dokumente der Teilmenge im Dokument enthalten sind. Es spielt dabei keine Rolle, welcher der beiden Ansätze zum Finden der Features verwendet wird. Wie beim Klassifizieren werden auch in den Vektoren für das Clustern die IDs dieser Features gespeichert, da nur dargestellt wird, welche Features enthalten sind. Die nicht-enthaltenen Features werden nicht

explizit gespeichert. Der Ablauf der Erstellung der Vektoren ist derselbe, wie in Kapitel 4.1.7.8 der vorliegenden Arbeit beschrieben.

5.1.2.6 Änderung des Vektorformats für das Clustern mit Text-Suffix-Fragment-Features

Da das Clustern mit Hilfe des Frameworks Natural Language Toolkit (NLTK)[1] durchgeführt wird und dieses in der Programmiersprache Python geschrieben ist, kann das Format der erstellten Vektoren nicht verwendet werden. Für das Clustern müssen die Vektoren der Daten in das Eingabeformat der Software umgewandelt werden. Dazu wird die Operation `changeVectorFormatToPython` aufgerufen. Sie liest die erzeugte Vektordatei mit allen Dokumenten der Teilmenge der gewünschten Klassenfamilie ein. Bei der Iteration über alle Vektoren wird die ID des Dokuments in einer Datei gespeichert, gefolgt von einem doppelten Leerzeichen und allen Feature-IDs des entsprechenden Dokuments aus der Vektordatei. Nach jeder Feature-ID stehen ein Doppelpunkt und eine Eins. Anstatt der Eins kann auch ein Gewicht angegeben werden. Getrennt werden die Feature-IDs durch ein Leerzeichen. Abschließend werden alle Vektoren der Daten in eine gemeinsame Datei geschrieben, die dann mit dem Python-Framework weiterverarbeitet werden kann.

5.1.2.7 Speichern der Klassen der Dokumente für das Clustern mit Text-Suffix-Fragment-Features

Um später die Evaluation durchführen zu können, muss rekonstruierbar sein, welches Dokument zu welcher Klasse gehört. Da nur die Dokument-ID und die Feature-IDs für das Clustern benötigt werden, wird eine zusätzliche Datei erzeugt, die die ID des Dokuments und die dazugehörige Klasse des Dokuments speichert.[2] Abschließend wird die so erzeugte Datei gespeichert.

5.1.3 Clustern mit Text-Suffix-Fragment-Features

5.1.3.1 Kurzeinführung Natural-Language-Toolkit-Framework

Die Algorithmen für das Erstellen des Clusterings - k-Means und Group-average-agglomerative-Clustering - stammen aus dem Natural Language Toolkit (NLTK).[3] Bei diesem Toolkit handelt es sich um eine Reihe von Modulen, die in der Programmiersprache Python geschrieben wurden und dazu dienen, möglichst viele Funktio-

1 Vgl. Bird u.a. (2009).
2 Anmerkung: Beim Clustern werden nur Dokumente mit genau einer Klassenzuordnung verwendet.
3 Vgl. Bird u.a. (2009).

nen, die zur Verarbeitung von natürlicher Sprache benötigt werden, zur Verfügung zu stellen. Es eignet sich insbesondere zur effektiven Erzeugung eines Prototyps, um bestimmte Verarbeitungsmöglichkeiten schnell testen zu können.[1] Genau aus diesem Grund wird es hier eingesetzt.
Dabei kommt lediglich ein Modul des NLTK zum Einsatz: das zum Erzeugen von Clusterings. Es enthält neben allgemeinen Definitionen einen abstrakten Clusterer und die beiden verwendeten Clusteralgorithmen. Der abstrakte Clusterer dient dazu, die gefundenen Features in einen Vektorraum zu überführen. Mit seiner Hilfe lässt sich auch die Dimensionalität dieser Vektoren verkleinern, das wird hier jedoch nicht angewendet. Zusätzlich stellt dieser Clusterer die Definition eines Dendrogramms zur Verfügung, also einer Datenstruktur, die ein hierarchisches Clustering optisch aufbereitet ausgeben kann. Die verwendeten Clusteralgorithmen entsprechen den theoretisch definierten in Kapitel 2.4.3. Zusätzlich sind von der Verfasserin einige Operationen programmiert worden, um das Modul des Toolkits an die benötigten Gegebenheiten anzupassen. Diese Anpassungen werden in den nachfolgenden Kapiteln erläutert.

5.1.3.2 Durchführen des Clusterns mit Text-Suffix-Fragment-Features

5.1.3.2.1 Clustern mit Text-Suffix-Fragment-Features mit dem k-Means-Clusterer

Das Clustern mit dem k-Means Clusterer beginnt mit dem Aufruf der Operation `doKMeansClustering`.[2] Diese Operation kann sowohl für die TSF-Feature-Vektoren als auch für die einzelwortbasierten Feature-Vektoren verwendet werden. In diesem Kapitel wird nur die Verwendung für die TSF-Feature-Vektoren beschrieben. Die davon abweichenden Schritte werden im Kapitel 5.2.3 erläutert.
Im Fall der TSF-Feature-Vektoren wird zunächst die Operation `readfileReuters-TrainVecSuffix` aufgerufen. Sie liest die Vektordatei ein und speichert die benötigten Informationen - overallIDs und dazugehörige Feature-IDs - in jeweils einer eigenen Datenstruktur. Anschließend erfolgt der Aufruf der Operation `readClassesOf-Doks`. Wie der Name bereits sagt, wird innerhalb der Operation die Datei mit den zu den Dokumenten gehörenden Klassen eingelesen und die Informationen - die overallID und die Klassen-ID - werden in einer Datenstruktur kombiniert gespeichert.

1 Vgl. Bird (2006), S. 69.
2 Anmerkung: Um die zusätzlichen Operationen von den vordefinierten zu unterscheiden, wird in der vorliegenden Arbeit eine Operation des NLTK in folgender Schreibweise verwendet: `nltk.Operationsname`, während die von der Verfasserin programmierten Operationen wie folgt geschrieben werden: `Operationsname`.

Nach diesen Vorbereitungen erfolgt der Aufruf der Operation `doClusteringPasses-Suffix`, die das eigentliche Clustern anstößt. Zu sehen ist der Ablauf der Operation `doKMeansClustering` in Abbildung 5.5.

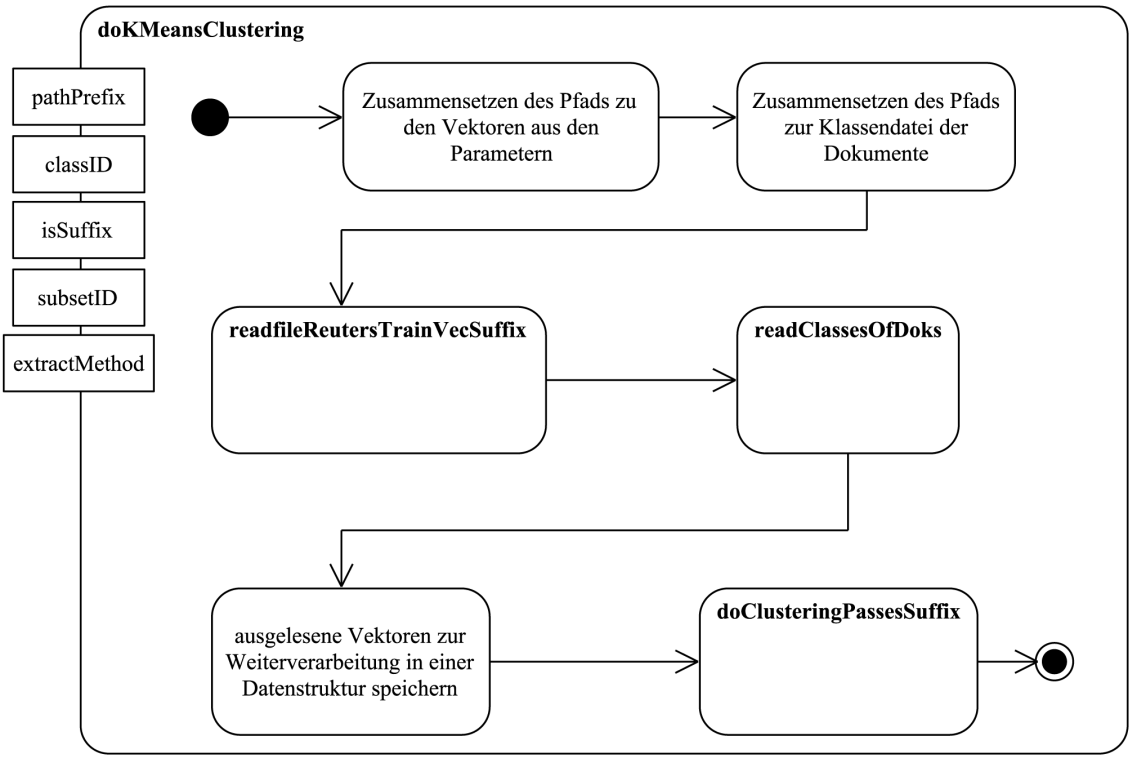


Abbildung 5.5: Ablauf der Operation `doKMeansClustering`

Die Operation `doClusteringPassesSuffix` stößt das eigentliche Clustern an. Zusätzlich wird in ihr gesteuert, mit welchem Distanzmaß das Clustern durchgeführt wird und wie viele Durchläufe es gibt. In der vorliegenden Arbeit werden die zwei vorgestellten Distanzmaße im k-Means verwendet.[1] Zusätzlich erfolgen für jedes Distanzmaß drei Durchläufe. Das geschieht, weil die Clusterzentren zunächst randomisiert ausgewählt werden und durch die drei[2] Durchläufe eine evtl. falsche Wahl ausgeglichen werden kann. In der Operation werden deshalb zwei Variablen angelegt: `algo` und `durchlauf`. Die erste steuert das zu verwendende Distanzmaß und die zweite die Durchläufe.
Es folgen zwei Schleifen. Die erste überprüft, ob bereits mit beiden Distanzmaßen geclustert wurde. Ist das der Fall, so endet die Operation. Ist es nicht der Fall, so

1 Siehe Kapitel 2.2.2 der vorliegenden Arbeit.
2 Die Anzahl von drei Durchläufen wurde von der Verfasserin gewählt, da drei Durchläufe zum einen in einer Tabelle darstellbar sind, was bei mehr Durchläufen schwieriger wird, und zum anderen keine „Patt“-Situation entstehen kann.

wird in einer zweiten Schleife überprüft, ob bereits drei Durchläufe durchgeführt wurden. Ist das der Fall, wird die Variable `algo` um Eins erhöht und die Variable `durchlauf` auf Null gesetzt, so dass die äußere Schleife wieder ausgeführt werden kann. Soll dagegen noch ein weiterer Durchlauf durchgeführt werden, wird zunächst unterschieden, welches Distanzmaß verwendet werden soll. Ist `algo` gleich Null, so wird mit der euklidischen Distanz[1] geclustert. Dafür wird ein Objekt der Klasse `nltk.KMeansClusterer` erzeugt und diesem als Distanzfunktion die euklidische Distanz übergeben. Im anderen Fall - `algo` ist gleich Eins - wird mit der Cosinus-Distanz[2] geclustert. Dann wird ebenfalls ein Objekt der Klasse `nltk.KMeansClusterer` erzeugt und diesem als Distanzfunktion die Cosinus-Distanz übergeben. In beiden Fällen wird der `durchlauf` um eins erhöht.

Anschließend wird die Operation `clusterSuffix` aufgerufen. Sie basiert auf der Operation `nltk.cluster` und verändert diese nur geringfügig, um mit dem anderen Vektorformat arbeiten zu können.[3] Die Operation `clusterSuffix` normalisiert die Vektoren und ruft die Operation `cluster_vectorspaceSuffix` auf, die auf der Operation `nltk.cluster_vectorspace` basiert. Abschließend werden die Vektoren über den Aufruf der Operation `classifySuffix`, die auf `nltk.classify` basiert, den Clustern zugeordnet.

Die Operation `cluster_vectorspaceSuffix` wählt durch Zufall aus den vorhandenen Vektoren Clustermittelpunkte in der gewünschten Anzahl aus.[4] Das eigentliche Clustern wird innerhalb der Operation `_cluster_vectorspaceSuffix`, die zum erzeugten Objekt der Klasse `nltk.KMeansClusterer` gehört und auf der Operation `nltk._cluster_vectorspace` basiert, durchgeführt. In dieser Operation werden alle Vektoren aufgrund der zu verwendenden Distanzfunktion dem Cluster zugeordnet, dessen Clustermittelpunkt sie am nächsten liegen. Anschließend werden neue Clustermittelpunkte berechnet. Die Clustermittelpunkte werden also nicht randomisiert ausgewählt wie im ersten Schritt, sondern aufgrund der in den Clustern vorhandenen Vektoren berechnet. Das geschieht, indem der Mittelwert der Vektoren ermittelt wird. Dieser Vektor ist der neue Clustermittelpunkt.

1 Siehe Kapitel 2.2.2.2 der vorliegenden Arbeit.
2 Siehe Kapitel 2.2.2 der vorliegenden Arbeit. Es handelt sich um die Euklid-Distanz und die Cosinus-Distanz.
3 Für die TSF-Feature-Vektoren wird ein anderes Format zur Darstellung der Vektoren in Python benötigt als ursprünglich im NLTK vorgesehen. Das liegt daran, dass die TSF-Feature-Vektoren eine sehr viel höhere Dimension aufweisen als von den Autoren des NLTK vorgesehen. Daher reicht das vom NLTK in allen Operationen verwendete Format zur Darstellung dieser großen Vektoren nicht aus. Daher werden die Operationen formatspezifisch angepasst, aber nicht in Bezug auf ihren Ablauf.
4 Die Parameter der in der Arbeit durchgeführten Experimente können in Kapitel 5.3 nachgelesen werden.

Um das k-Means-Clustern zu beenden, existieren mehrere Möglichkeiten, siehe Kapitel 2.4.3.1. In der vorliegenden Arbeit wird der im NLTK festgelegte Schwellenwert verwendet, unter dem die Abweichung zwischen den alten Clustermittelpunkten und den neu berechneten liegen muss.

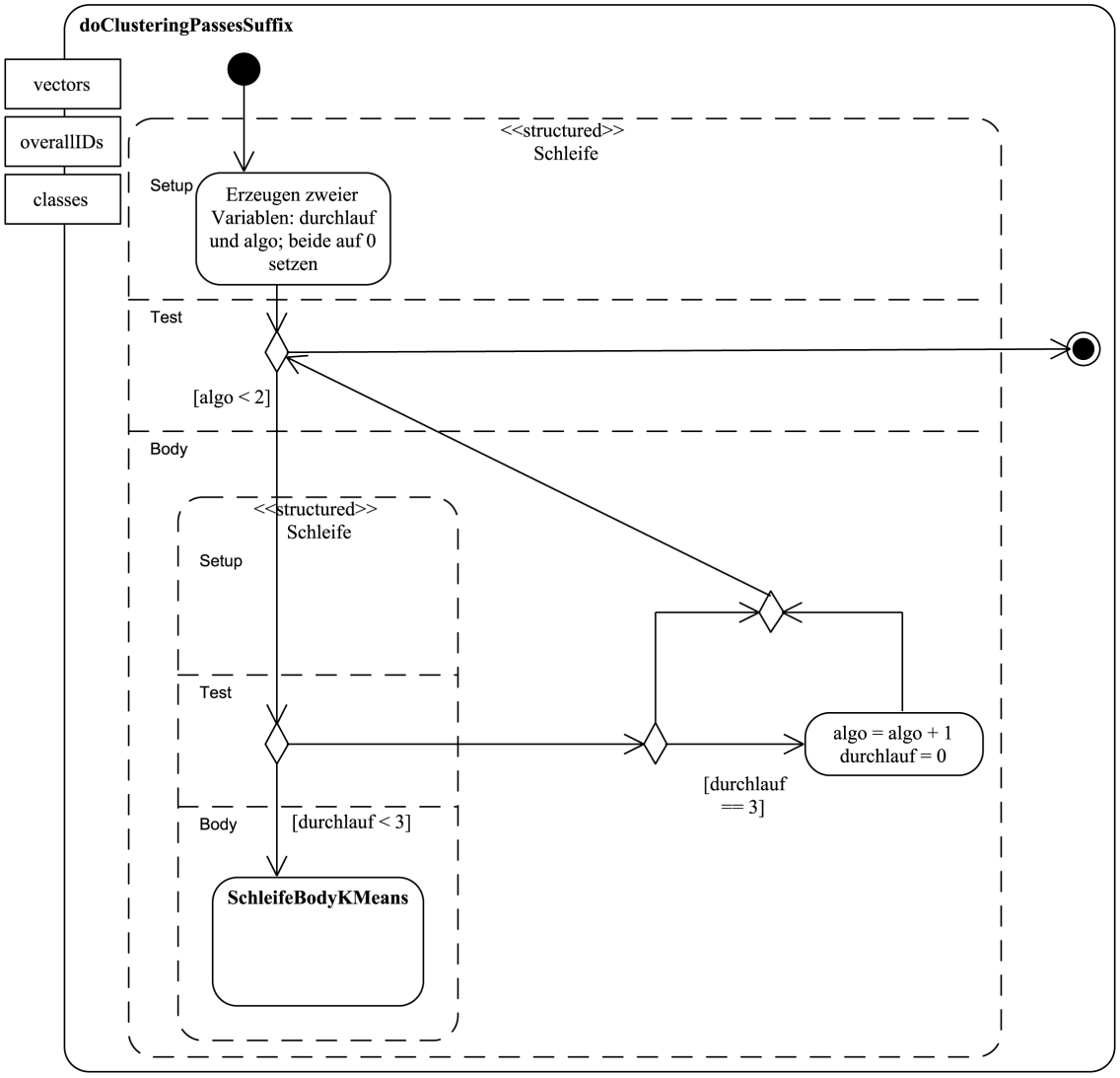


Abbildung 5.6: Ablauf der Operation `doClusteringPassesSuffix`

Es wird also die Differenz zwischen den Clustermittelpunkten der Runde zuvor und den neu berechneten ermittelt. Ist diese Differenz kleiner als der Schwellenwert[1], so muss kein erneutes Clustern stattfinden. In diesem Fall werden die Clustermittelpunkte auf die neu berechneten gesetzt und die Operation wird beendet. Im ande-

1 Der in den Experimenten verwendete Schwellenwert kann dort nachgelesen werden, siehe Kapitel 5.3.

ren Fall werden wieder die Vektoren aufgrund des Distanzmaßes dem Cluster mit der kleinsten Distanz zwischen Vektor und Clustermittelpunkt zugeordnet und die Clustermittelpunkte neu berechnet. Das geschieht so lange, bis der Schwellenwert unterschritten wird.

Damit ist das Clustern eines Durchlaufs in der Operation `doClusteringPasses-Suffix` abgeschlossen und es kann evaluiert werden. Die Erläuterung der Operation `evaluateClusteringKMeans` befindet sich in Kapitel 5.1.4.1. Die Darstellung der Operation ist in den Abbildungen 5.6 auf S. 468 und 5.7 zu sehen.

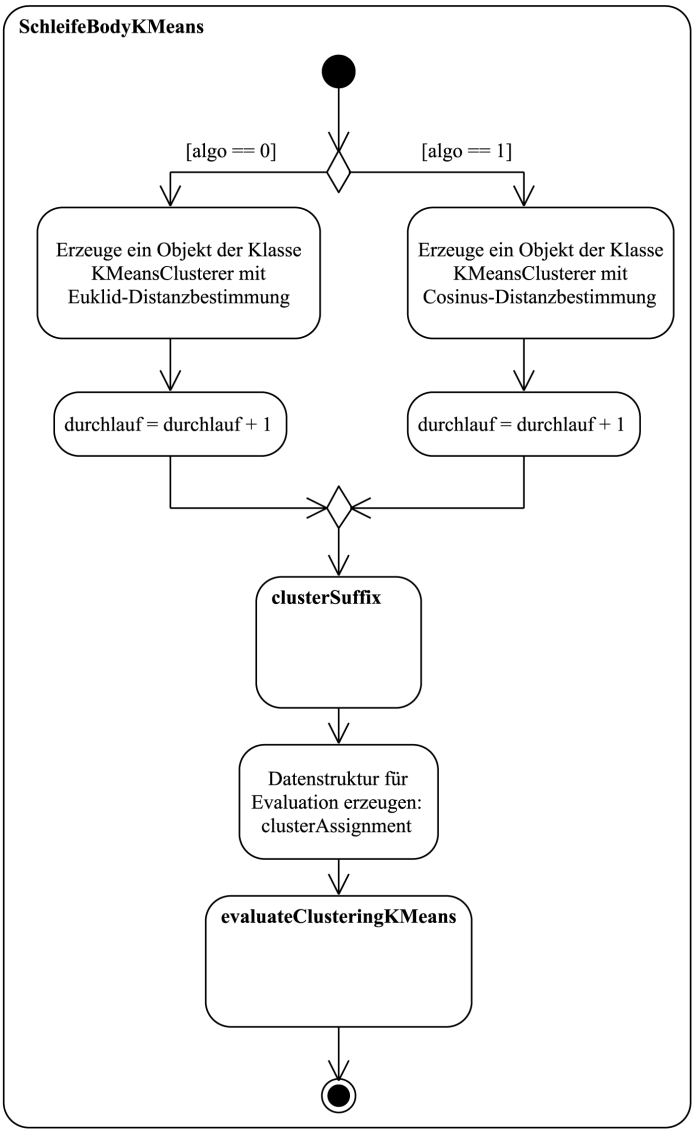


Abbildung 5.7: Ablauf der Schleife der Operation `doClusteringPassesSuffix`

5.1.3.2.2 Clustern mit Text-Suffix-Fragment-Features mit dem Group-average-agglomerative-clustering-Clusterer

Das Clustern mit dem GAAC-Clusterer beginnt mit dem Aufruf der Operation `doGAACClustering`. Diese Operation kann sowohl für die TSF-Feature-vektoren als auch für die einzelwortbasierten Feature-Vektoren verwendet werden. In diesem Kapitel wird nur die Verwendung für die TSF-Feature-Vektoren beschrieben und die davon abweichenden Schritte werden im Kapitel 5.2.3 erläutert.

Die Operation beginnt genauso wie die Operation `doKMeansClustering`. Das Einlesen der Vektoren und Klassen erfolgt über die gleichen Operationen. Der Unterschied besteht im anschließenden Erzeugen des Objekts, mit dessen Operationen das Clustern durchgeführt wird. In der Operation `doGAACClustering` wird ein Objekt der Klasse `nltk.GAACClusterer` erzeugt und anschließend die bereits bekannte Operation `clusterSuffix` aufgerufen. Diese ist die gleiche, wie beim k-Means-Clusterer beschrieben. Das heißt, der Ablauf ist genau der gleiche bis auf einen Unterschied: Wenn `_cluster_vectorspaceSuffix` aufgerufen wird, erfolgt dies jetzt für das Objekt der Klasse `nltk.GAACClusterer` und nicht wie zuvor für den k-Means-Clusterer.

Der GAAC-Clusterer betrachtet zunächst alle Vektoren als einzelne Cluster und fügt Schritt für Schritt diejenigen Cluster zusammen, die die größte Ähnlichkeit zueinander haben. Die Ähnlichkeitsberechnung erfolgt wie in Kapitel 2.4.3.2 beschrieben. Das Abbruchkriterium ist in der vorliegenden Arbeit die Anzahl der gewünschten Cluster, die zuvor festgelegt wird.[1] Wird keine Anzahl festgelegt, so werden alle Cluster schließlich zu einem verschmolzen. Das zuvor erwähnte Dendrogramm dient der Speicherung des Clusterings in einer Baumform, die die Reihenfolge abbildet, in der die Cluster verschmolzen werden. Ist das Clustern bis zum Abbruchkriterium durchgeführt, enthält das Dendrogramm alle benötigten Informationen, um die Qualität beurteilen zu können. Dafür wird es in der Operation `doGAACClustering` ausgelesen und so aufbereitet, dass die Evaluierungsoperation damit arbeiten kann. Diese wird abschließend aufgerufen.[2] Dargestellt ist die Operation `doGAACClustering` in Abbildung 5.8.

1 Siehe Kapitel 5.3 für die Anzahl der Cluster der Experimente.
2 Eine Erläuterung dieser Operation befindet sich in Kapitel 5.1.4.1.

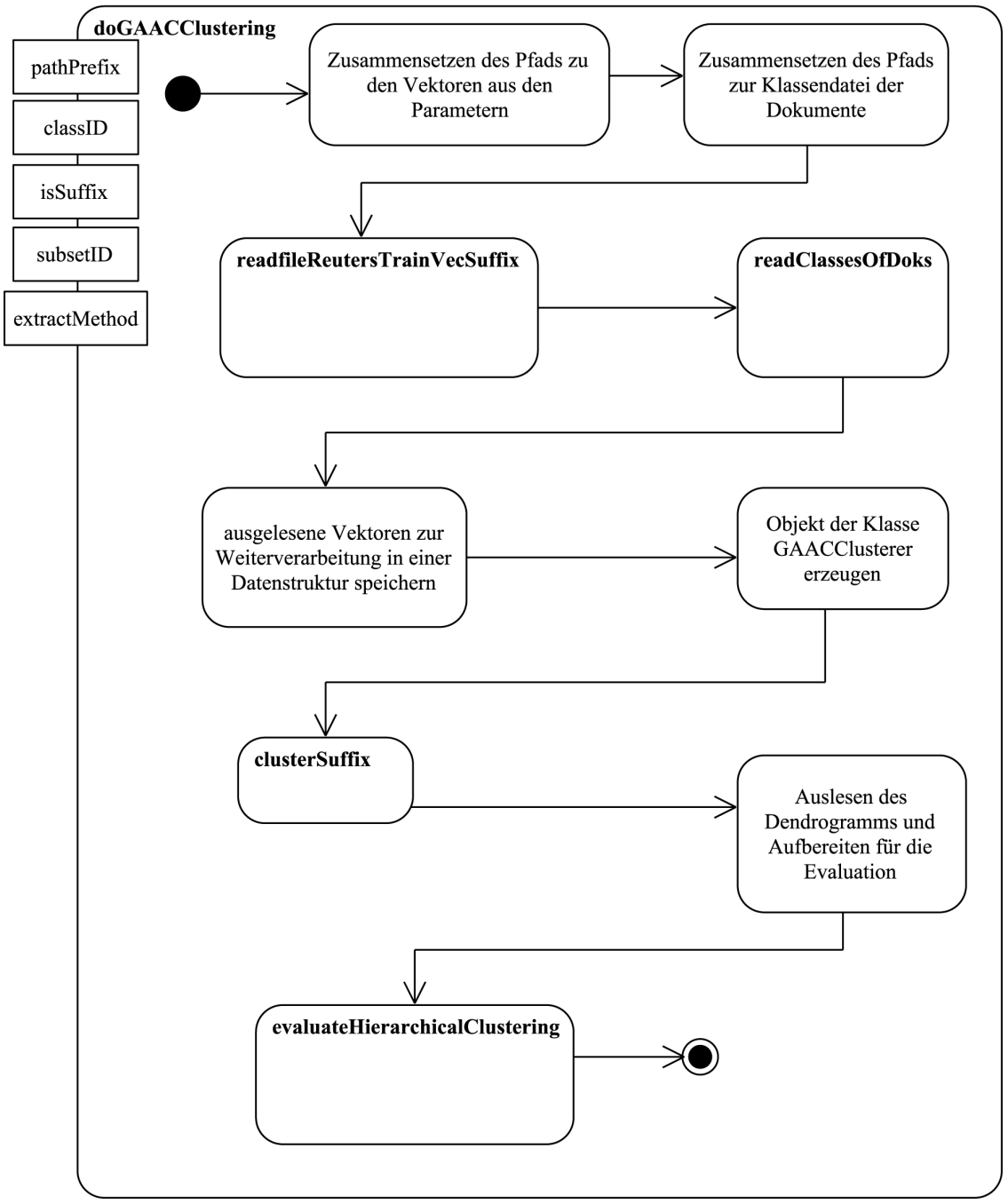


Abbildung 5.8: Ablauf der Operation `doGAACClustering`

5.1.4 Evaluation des Clusterns mit Text-Suffix-Fragment-Features

5.1.4.1 Vorbereitung der Daten für die Evaluation

Die zwei bereits in Kapitel 5.1.3.2 der vorliegenden Arbeit genannten Operationen `evaluateClusteringKMeans` und `evaluateHierarchicalClustering` stoßen

die Evaluierung der erzeugten Clusterings an. Bevor eine Evaluation der Ergebnisse möglich ist, müssen diese jedoch in eine Form gebracht werden, in der sie von den Evaluationsoperationen verarbeitet werden können.
Für alle Operationen, die die eigentliche Evaluation durchführen, sind insbesondere die Angabe der TP, FP, FN und TN wichtig. Diese Anzahlen müssen also bestimmt werden. Zusätzlich ist es für die Berechnung der Purity entscheidend, dass die maximale Precision pro Cluster berechnet ist.
Die ersten Schritte innerhalb der Operation `evaluateClusteringKMeans` dienen dazu, die Clusteringdaten so aufzubereiten, dass sowohl die TP, FP, FN und TN über das gesamte Clustering, als auch die maximale Precision pro Cluster gezählt bzw. berechnet werden können. Dafür werden die Informationen, welcher Vektor zu welchem Cluster gehört, wie folgt aufbereitet:

- Die Kombination `clusterID` und Liste der dazugehörigen Kombinationen aus overallID und Klasse zeigt die Zugehörigkeit der Dokumente mit ihren von Reuters zugewiesenen Klassen zum jeweils erzeugten Cluster an.

- Die Kombination aus ID der Klasse und Liste der dazugehörigen Dokumente zeigt den gleichen Sachverhalt, also die Zugehörigkeit der Dokumente zu den von Reuters zugewiesenen Klassen an, jedoch so, dass über die Klassen-ID auf die Dokumente zugegriffen werden kann. Dies wird für jedes Cluster einzeln bestimmt.

Mit Hilfe dieser beiden Kombinationen kann innerhalb der Operation `countTruePositives` die Anzahl der *true positives* pro Cluster berechnet werden und zusätzlich die Anzahl der Dokumente pro Klasse über das gesamte erzeugte Clustering. Dafür wird zunächst aus dem gerade betrachteten Cluster die Anzahl der Dokumente der gerade betrachteten Klasse, von der Dokumente im Cluster vorhanden sind, ausgelesen. Unter Verwendung der Formel (a) auf Seite 74 wird die TP dieser Klasse in diesem Cluster berechnet und in einer Liste für alle Anzahlen von *true positives* dieses Clusters gespeichert. Anschließend wird die Anzahl der Dokumente dieser Klasse in einer Datenstruktur als Wert gespeichert und die Klassen-ID als Schlüssel. Wurde die Klassen-ID bereits gespeichert, so wird die vorhandene Anzahl der Dokumente um die Anzahl der im gerade betrachteten Cluster vorhandenen Dokumente erhöht. Sind alle im Cluster vorhandenen Klassen auf diese Weise verarbeitet worden, so wird die Liste der TP des Clusters und die Zuordnung der pro Klasse im Cluster vorhandenen Dokumente zurückgegeben.
Innerhalb der Operation `evaluateClusteringKMeans` wird zusätzlich in einer Datenstruktur gespeichert, wie viele Dokumente insgesamt, also nicht nach Klassen

getrennt, im Cluster vorhanden sind. Mit einer Schleife über die Liste der TP im gerade betrachteten Cluster, in dem die einzelnen TP pro Klasse ausgelesen und aufsummiert werden, wird die Gesamtanzahl an TP für das Cluster ermittelt. Sind alle Cluster des Clusterings auf diese Weise verarbeitet worden, so ist die Gesamtanzahl der *true positives* des Clusterings bestimmt.

Mit diesen bereits ermittelten Informationen, also den TP des Clusterings sowie der Kombination aus vorhandenen Dokumenten pro Klasse und pro Cluster können die FN, FP und TN ermittelt werden. Das erfolgt durch den Aufruf der Operationen `computeFalseNegatives`, `computeFalsePositives` bzw. `computeTrueNegatives`. Innerhalb der Operationen werden die entsprechenden Anzahlen wie in den Formeln (b) bis (d) auf S. 74 berechnet und zurückgegeben.

Um später die Purity berechnen zu können, muss noch die Berechnung der maximalen Precision pro Cluster erfolgen. In einer weiteren Schleife über die im Clustering vorhandenen Cluster erfolgt der Aufruf der Operation `computePrecisionsForCluster`. Innerhalb der Operation werden alle im gerade betrachteten Cluster vorhandenen Klassen betrachtet und für sie, nach Formel 2.15 aus Kapitel 2.5.3.2.2, die Precision berechnet. Alle berechneten Precisions des Clusters werden gespeichert und zurückgegeben. In der Operation `evaluateClusteringKMeans` wird diese Liste von Precisions des Clusters sortiert und die höchste Precision ausgelesen. Diese wird zusammen mit der `clusterID` gespeichert, so dass später die Purity berechnet werden kann, siehe Kapitel 5.1.4.2.

Mit dieser Berechnung ist die Vorbereitung der Daten für das k-Means-Clustering abgeschlossen und innerhalb der Operation `evaluateClusteringKMeans` wird mit den Berechnungen der verwendeten Evaluationsmaße fortgefahren. Das beinhaltet den Aufruf der in Kapitel 5.1.4.2 bis 5.1.4.4 beschriebenen Operationen und die Ausgabe der berechneten Werte.

In der Operation `evaluateHierarchicalClustering` müssen ebenfalls die vorhandenen Informationen aus dem erzeugten Clustering so aufbereitet werden, dass die Operationen zur Berechnung der verwendeten Evaluationsmaße aufgerufen werden können. Das beinhaltet im Fall des hierarchischen Clusterings nicht nur das Erzeugen der benötigten Kombinationen, sondern bevor das durchgeführt werden kann, muss das erzeugte Dendrogramm so verarbeitet werden, dass die benötigten Informationen ermittelt werden können.

Beim partitionierenden Clustering werden als Ergebnis die erzeugten Cluster mit den dazugehörigen Dokumenten zurückgegeben. Im Fall des hierarchischen Clusterings ist die Rückgabe ein Dendrogramm, d.h., die erzeugte Anzahl an Clustern, die jedoch nicht in jedem Fall die Dokumente beinhalten, sondern auch oder nur andere

Cluster, aus denen die jeweils in der Hierarchie darüberliegenden Cluster erzeugt wurden. Um an die zur Evaluation benötigten Informationen zu kommen, muss also zunächst die hierarchische Struktur aufgelöst werden, um die Zuordnung der Dokumente zu den Clustern der obersten Hierarchieebene zu erhalten. Das erfolgt, indem die Baumstruktur von der Wurzel[1], also den Clustern der obersten Hierarchieebene, bis zu den Blättern, also den Dokumenten, durchlaufen wird und jeweils der `clusterID` der obersten Hierarchieebene die dazugehörigen Dokumenten-IDs zugeordnet werden. Ist diese Kombination vorhanden, so kann pro Cluster ermittelt werden, welchen Klassen die jeweils im Cluster vorhandenen Dokumente angehören, und diese Kombination ebenfalls gespeichert werden. Ist das abgeschlossen, so sind genau die gleichen Informationen wie im Fall der Evaluation des k-Means-Clusterings vorhanden.
Ab diesem Punkt innerhalb der Operation `evaluateHierarchicalClustering` werden demnach die gleichen Operationen in der gleichen Reihenfolge aufgerufen wie in der Operation `evaluateKMeansClustering`, beginnend mit der Operation `countTruePositives`.

5.1.4.2 Evaluation mit der Purity

Die Purity selbst wird wie in Kapitel 2.5.3.2.2 beschrieben berechnet. Wie zuvor erläutert, kann man folgende Daten an die Operation `computeWeightedPurity` übergeben:

- Die Kombination aus `clusterID` und der maximalen Precision des entsprechenden Clusters, um die gewichtete Purity über alle Cluster bestimmen zu können.
- Die Kombination aus `clusterID` und der Anzahl der im Cluster vorhandenen Dokumente, um das ebenfalls bei der Berechnung verwenden zu können.

Zunächst wird die Gesamtanzahl der Dokumente über alle Cluster bestimmt, da dieser Wert für die Gewichtung der Purity benötigt wird. Das erfolgt über eine Schleife über die Kombination aus `clusterID` und der Anzahl der Dokumente im Cluster, wobei die Anzahl jeweils aufsummiert wird. Anschließend erfolgt in einer weiteren Schleife die Berechnung wie in Formel 2.16 in Kapitel 2.5.3.2.2 angegeben, wobei jeweils die maximale Precision des gerade betrachteten Clusters einfließt und der Wert zur insgesamt bisher berechneten gewichteten Purity hinzuaddiert wird. Abschließend kann die gewichtete Purity zurück- und ausgegeben werden.

1 Anmerkung: Soll mehr als ein Cluster erzeugt werden, so existieren so viele Wurzeln, wie Cluster erzeugt werden sollen.

5.1.4.3 Evaluation mit dem Rand Index

Der Rand Index wird in der Implementierung, wie in Kapitel 2.5.3.2.1 beschrieben, berechnet. Benötigt werden dafür die TP, FP, FN und TN über alle erzeugten Cluster. Diese sind, wie zuvor beschrieben, bereits berechnet worden. Der Ablauf der Operation `computeRandIndex` umfasst deshalb das Berechnen des Rand Index nach Formel 2.14 aus Kapitel 2.5.3.2.1 sowie die Rück- und Ausgabe des Wertes für das erzeugte Clustering.

5.1.4.4 Evaluation mit dem F-Maß für das Clustern

Die Berechnung des F_β-Maßes erfolgt, wie in Kapitel 2.5.3.2.3 beschrieben. Der Aufruf der Operation `computeFMeasure` ist mit der Übergabe der TP, FP und FN verbunden, die, wie zuvor beschrieben, für das gesamte Clustering berechnet wurden. Zusätzlich wird der Parameter `beta` übergeben, der die Gewichtung des F_β-Maßes wie in Kapitel 2.5.2.2 erläutert festlegt.[1] Innerhalb der Operation `computeFMeasure` wird zunächst die Precision berechnet und anschließend der Recall. Abschließend wird aus diesen beiden Komponenten und `beta` das entsprechende F_β-Maß berechnet, zurückgegeben und ausgegeben.

5.2 Implementierung des Clusterns mit Einzelwort-Features

5.2.1 Einleitung der Implementierung des Clusterns mit Einzelwort-Features

Um vergleichbare Ergebnisse zu erzielen, müssen die gleichen Verarbeitungsschritte mit den Lewis-Feature-Vektoren durchgeführt werden, die bei den auf TSF-Features basierenden Vektoren durchgeführt wurden. Hier ist jedoch der Vorteil gegeben, dass die Vektoren bereits vorliegen. Das bedeutet, sie müssen nicht erzeugt werden, sondern lediglich in eine Form gebracht werden, die den auf TSF-Features basierenden Vektoren entspricht, um sie dann in den Algorithmen zum Clustern verwenden zu können.
Insgesamt bedeutet das also, dass die ersten Schritte vom Erstellen der Teilmengen bis einschließlich zum Erstellen der Vektoren hier entfallen und nur eine Beschreibung der Umwandlung des Vektorformats sowie des Clusterns und der Evaluation erfolgt.

1 Die Belegung dieses Parameters in den Experimenten kann in Kapitel 5.3 nachgelesen werden.

5.2.2 Daten vorbereiten für das Clustern mit Einzelwort-Features

5.2.2.1 Änderung des Vektorformats für das Clustern mit Einzelwort-Features

Wie bereits in Kapitel 5.1.2 erläutert, wird zum Clustern das Natural Language Toolkit verwendet. Aus diesem Grund müssen die Lewis-Feature-Vektoren in ein Format umgewandelt werden, das dem Eingabeformat der verwendeten Software entspricht. Das erfolgt auf die gleiche Weise, wie in Kapitel 5.1.2.6 beschrieben. Der einzige Unterschied besteht darin, dass die Operation anders heißt: `changeVectorFormatToPythonLewis`. Der andere Name beinhaltet andere Pfadbezeichnungen innerhalb der Operation, so dass auf die von Lewis zur Verfügung gestellten Vektoren zugegriffen wird anstatt auf die Vektoren mit den TSF-Features.

5.2.2.2 Speichern der Klassen der Dokumente für das Clustern mit Einzelwort-Features

Dieser Schritt entfällt für die Lewis-Feature-Vektoren ebenfalls, da er bereits für die TSF-Feature-Vektoren durchgeführt wurde und die gleichen Vektoren im einzelwortbasierten Fall ebenfalls verwendet werden. Die Klassen der Dokumente sind also bereits bekannt und müssen nicht nochmals ausgelesen werden.

5.2.3 Clustern mit Einzelwort-Features

5.2.3.1 Clustern mit Einzelwort-Features mit dem k-Means-Clusterer

Für die einzelwortbasierten Feature-Vektoren erfolgt die gleiche Verarbeitung beim Clustern mit k-Means wie für die TSF-Feature-Vektoren, siehe Kapitel 5.1.3.2.1. Es erfolgt ebenfalls der Aufruf der Operation `doKMeansClustering`, in der dann die entsprechenden Verarbeitungsschritte durchgeführt werden.

5.2.3.2 Clustern mit Einzelwort-Features mit dem Group-average-agglomerative-clustering-Clusterer

Für die einzelwortbasierten Feature-Vektoren erfolgt die gleiche Verarbeitung beim Clustern mit dem GAAC-Clusterer wie für die TSF-Feature-Vektoren, siehe Kapitel 5.1.3.2.2. Es erfolgt ebenfalls der Aufruf der Operationen `doGAACClustering` und `doKMeansClustering`, in denen dann die entsprechenden Verarbeitungsschritte durchgeführt werden.

5.2.4 Evaluation des Clusterns mit Einzelwort-Features

Die Evaluation im Fall der einzelwortbasierten Vektoren läuft genauso ab, wie im Kapitel 5.1.4 beschrieben. Es werden also ebenfalls die Operationen `evaluateClusteringKMeans` und `evaluateHierarchicalClustering` aufgerufen.

5.3 Durchgeführte Clusterexperimente

5.3.1 Definition eines Clusterexperiments

Im Rahmen dieser Arbeit wird das entwickelte Verfahren der Ermittlung von TSF-Features für textuelle Dokumente über Suffix Arrays mit Hilfe von Experimenten überprüft. Dabei wird das gleiche Experiment zum einen für die mit dem entwickelten Verfahren erzeugten TSF-Features und zum anderen mit den Lewis-Feature-Vektoren durchgeführt und die Ergebnisse werden miteinander verglichen. Für jedes Experiment erfolgt eine Beschreibung nach dem in Kapitel 4.3.1 aufgestellten Schema. Einige Punkte weichen von den dort definierten ab, diese werden nachfolgend erläutert.

- Daten

Da beim Clustern keine Aufteilung in Trainings- und Testdaten erfolgt, werden die Punkte „Trainingsdaten“ und „Testdaten“ zu einem gemeinsamen Punkt „Daten“ zusammengeführt. Es wird beschrieben, welche Daten für das Clustern verwendet werden. Darunter fallen die Anzahl der Teilmengen, ihre Bezeichnungen, die Anzahl der jeweils enthaltenen Dokumente und die Angabe, zu wie vielen verschiedenen Klassen die Dokumente gehören. So würde man beispielsweise angeben, dass man eine Teilmenge mit der Bezeichnung T1 und darin enthaltenen 20 Dokumenten, die zu 2 verschiedenen Region-Klassen gehören, clustert.

- Anzahl Cluster

Aus dem vorherigen Punkt, also aus der Angabe, zu wie vielen verschiedenen Klassen die verwendeten Dokumente gehören, ergibt sich automatisch die Anzahl der zu erzeugenden Cluster. Unter diesem Punkt wird diese Anzahl noch einmal explizit angegeben.

- Algorithmen

Zusätzlich zu der Angabe, welcher Algorithmus für das Clustern verwendet wird, erfolgt unter diesem Punkt die Angabe, welches Distanzmaß verwendet wird. Zudem wird angegeben, wie viele Durchläufe jeweils vorgenommen werden, aus denen abschließend der beste Durchlauf bestimmt wird, um insgesamt zu einem Ergebnis zu gelangen.
Somit ergibt sich das folgende Schema für die Beschreibung eines Clusterexperiments[1]:

- ID
- Bezeichnung
- Daten
- Ähnlichkeit
- Anzahl Cluster
- Algorithmen
- Evaluation
- Ergebnisse
- Interpretation der Ergebnisse

5.3.2 Beschreibung des beim Clustern verwendeten Datensatzes

Der beim Clustern verwendete Datensatz entspricht dem beim Klassifizieren verwendeten, siehe Kapitel 4.1.2. Da beim Clustern keine Unterscheidung in Trainings- und Testdaten vorgenommen wird, werden beim Clustern nur die Trainingsdaten des Datensatzes verwendet. Zusätzlich erfolgt das Clustern nur mit Dokumenten, die nur einer Klasse der Regions-Klassenfamilie zugeordnet sind. Das bedeutet, das Clustern erfolgt auch nur aufgrund der Region-Klassenfamilie und nicht aufgrund der anderen beiden Klassenfamilien.

1 Anmerkung: Zwar werden beim Clustern zwei unterschiedliche Arten von TSF-Feature-Vektoren verwendet, siehe Kapitel 5.1.2.3, jedoch werden die Ergebnisse beider Featurearten jeweils pro Experiment erfasst. Deshalb ist die Erzeugungsart der TSF-Feature-Vektoren kein eigener Unterpunkt.

Dafür existieren zwei Gründe:

1. Die Region-Klassenfamilie ist die einzige der drei Klassenfamilien, die keine Hierarchie hat. Das vereinfacht das Durchführen des Clusterns und die anschließende Auswertung der Ergebnisse.

2. Die Ergebnisse der Region-Klassenfamilie beim Klassifizieren sind in den durchgeführten Experimenten durchgehend für die wortübergreifenden Features besser als für die einzelwortbasierten. Das ist für die anderen beiden Klassenfamilien nicht der Fall.

5.3.3 Clusterexperimente

5.3.3.1 Experiment 7

- ID
  Clus1RV

- Bezeichnung
  vektorbasiertes Region-Clustern von Daten der Reuters-Daten RCV1-v2 mit Purity als Evaluationsmaß

- Daten
  - 3 randomisiert zusammengestellte Teilmengen der Trainingsdaten des RCV1-v2 mit jeweils 2.000 Dokumenten: T1, T2, T3
  - Dokumente von T1 bis T3 gehören zu genau einer der 4 Region-Klassen „USA“, „UK“, „JAP“ und „AUSTR“
  - Die 4 angegebenen Klassen sind die Klassen mit den meisten Dokumenten innerhalb der Trainingsdaten des RCV1-v2.
  - Um gleich große Cluster erreichen zu können, gehören jeweils 500 Dokumente zu einer der genannten Klassen.

- Ähnlichkeit
  vektorbasiert

- Algorithmen
  - k-Means mit Euklid als Distanzmaß, einem Schwellenwert von 10^{-6} und jeweils 3 Durchläufen pro Teilmenge, das beste Ergebnis wird gewertet

  - k-Means mit Cosinus als Distanzmaß, einem Schwellenwert von 10^{-6} und jeweils 3 Durchläufen pro Teilmenge, das beste Ergebnis wird gewertet
  - GAAC

- Anzahl Cluster
  4

- Evaluation
  gewichtete Purity

- Ergebnisse[1]
  Die Ergebnisse für den k-Means-Clusterer mit euklidischem Distanzmaß befinden sich in den Tabellen 5.1 bis 5.3 auf dieser Seite bis S. 481.
  Die Ergebnisse für den k-Means-Clusterer mit Cosinus-Distanzmaß befinden sich in den Tabellen 5.4 bis 5.6 auf den Seiten 481 bis 482.
  Die Ergebnisse für den GAAC-Clusterer befinden sich in Tabelle 5.7 auf S. 483.

Tabelle 5.1: Ergebnisse des Experiments Clus1RV für Datenmenge T1 und dem k-Means-Clusterer mit Euklid-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	Purity in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	58,55
	Region	1	„SingleDoks“	49,10
Lewis-Feature-Vektoren	Region	1	-	55,45
TSF-Feature-Vektoren	Region	2	„AllDoks“	39,20
	Region	2	„SingleDoks“	40,45
Lewis-Feature-Vektoren	Region	2	-	54,20
TSF-Feature-Vektoren	Region	3	„AllDoks“	39,45
	Region	3	„SingleDoks“	42,15
Lewis-Feature-Vektoren	Region	3	-	45,11

1 In allen Tabellen werden die Ergebnisse auf zwei Stellen nach dem Komma gerundet. Das jeweils dunkel hinterlegte Tabellenfeld kennzeichnet bei k-Means das beste Ergebnis über die drei Durchläufe pro Feature-Erzeugungsmethode.

Tabelle 5.2: Ergebnisse des Experiments Clus1RV für Datenmenge T2 und dem k-Means-Clusterer mit Euklid-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	Purity in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	44,40
	Region	1	„SingleDoks“	37,85
Lewis-Feature-Vektoren	Region	1	-	50,95
TSF-Feature-Vektoren	Region	2	„AllDoks“	41,50
	Region	2	„SingleDoks“	43,45
Lewis-Feature-Vektoren	Region	2	-	48,15
TSF-Feature-Vektoren	Region	3	„AllDoks“	54,10
	Region	3	„SingleDoks“	43,35
Lewis-Feature-Vektoren	Region	3	-	48,65

Tabelle 5.3: Ergebnisse des Experiments Clus1RV für Datenmenge T3 und dem k-Means-Clusterer mit Euklid-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	Purity in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	42,50
	Region	1	„SingleDoks“	39,70
Lewis-Feature-Vektoren	Region	1	-	42,00
TSF-Feature-Vektoren	Region	2	„AllDoks“	50,05
	Region	2	„SingleDoks“	54,65
Lewis-Feature-Vektoren	Region	2	-	43,25
TSF-Feature-Vektoren	Region	3	„AllDoks“	52,20
	Region	3	„SingleDoks“	38,35
Lewis-Feature-Vektoren	Region	3	-	43,60

Tabelle 5.4: Ergebnisse des Experiments Clus1RV für Datenmenge T1 und dem k-Means-Clusterer mit Cosinus-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	Purity in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	50,45
	Region	1	„SingleDoks“	40,90
Lewis-Feature-Vektoren	Region	1	-	47,05
wird auf der nächsten Seite fortgesetzt				

Tabelle 5.4: Ergebnisse des Experiments Clus1RV für Datenmenge T1 und dem k-Means-Clusterer mit Cosinus-Distanzmaß (Fortsetzung)

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	Purity in %
TSF-Feature-Vektoren	Region	2	„AllDoks“	35,50
	Region	2	„SingleDoks“	50,90
Lewis-Feature-Vektoren	Region	2	-	44,95
TSF-Feature-Vektoren	Region	3	„AllDoks“	51,75
	Region	3	„SingleDoks“	48,35
Lewis-Feature-Vektoren	Region	3	-	47,45

Tabelle 5.5: Ergebnisse des Experiments Clus1RV für Datenmenge T2 und dem k-Means-Clusterer mit Cosinus-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	Purity in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	42,45
	Region	1	„SingleDoks“	45,05
Lewis-Feature-Vektoren	Region	1	-	48,25
TSF-Feature-Vektoren	Region	2	„AllDoks“	43,90
	Region	2	„SingleDoks“	45,60
Lewis-Feature-Vektoren	Region	2	-	46,20
TSF-Feature-Vektoren	Region	3	„AllDoks“	39,85
	Region	3	„SingleDoks“	46,45
Lewis-Feature-Vektoren	Region	3	-	47,15

Tabelle 5.6: Ergebnisse des Experiments Clus1RV für Datenmenge T3 und dem k-Means-Clusterer mit Cosinus-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	Purity in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	39,75
	Region	1	„SingleDoks“	37,30
Lewis-Feature-Vektoren	Region	1	-	44,80
TSF-Feature-Vektoren	Region	2	„AllDoks“	47,50
	Region	2	„SingleDoks“	41,30
Lewis-Feature-Vektoren	Region	2	-	40,30

wird auf der nächsten Seite fortgesetzt

Tabelle 5.6: Ergebnisse des Experiments Clus1RV für Datenmenge T3 und dem k-Means-Clusterer mit Cosinus-Distanzmaß (Fortsetzung)

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	Purity in %
TSF-Feature-Vektoren	Region	3	„AllDoks“	40,30
	Region	3	„SingleDoks“	49,40
Lewis-Feature-Vektoren	Region	3	-	42,60

Tabelle 5.7: Ergebnisse des Experiments Clus1RV für Datenmenge T1 bis T3 und dem GAAC-Clusterer

Verfahren	Daten-menge	Klassen-familie	Methode TSF-Features	Purity in %
TSF-Feature-Vektoren	1	Region	„AllDoks“	43,15
	1	Region	„SingleDoks“	35,85
Lewis-Feature-Vektoren	1	Region	-	46,10
TSF-Feature-Vektoren	2	Region	„AllDoks“	38,50
	2	Region	„SingleDoks“	39,40
Lewis-Feature-Vektoren	2	Region	-	44,60
TSF-Feature-Vektoren	3	Region	„AllDoks“	45,65
	3	Region	„SingleDoks“	36,55
Lewis-Feature-Vektoren	3	Region	-	45,40

- Interpretation der Ergebnisse
  - Einführung
    Im Folgenden werden die vorgestellten Ergebnisse für die auf den Suffix Arrays basierenden Vektoren und für die Lewis-Feature-Vektoren miteinander verglichen. Dabei werden zunächst die besten Ergebnisse pro Algorithmus und Distanzmaß über die drei Datenmengen miteinander verglichen. Abschließend erfolgt ein Vergleich für das gesamte Experiment.
  - Ergebnisvergleich k-Means mit Euklid
    Bildet man die Ergebnisse, die sich durch die Auswertung des k-Means-Clusterers mit Euklid als Distanzmaß für die Purity für alle Datenmengen ergeben, in Diagrammen ab, so erhält man die Abbildung 5.9 auf S. 484. Im Fall der TSF-Feature-Erzeugung „AllDoks“ ergibt sich über die drei Datenmengen für den k-Means-Clusterer mit euklidischem Distanzmaß ein eindeutiges

Ergebnis, welches Verfahren besser ist. So erreichen die TSF-Feature-Vektoren für alle Datenmengen ein besseres Ergebnis für die Purity als die einzelwortbasierten Feature-Vektoren. Im Durchschnitt unterscheiden sich die Ergebnisse um gerundet 4,95 Prozentpunkte, wobei der minimale Abstand 3,1 Prozentpunkte und der maximale Abstand 8,6 Prozentpunkte beträgt.

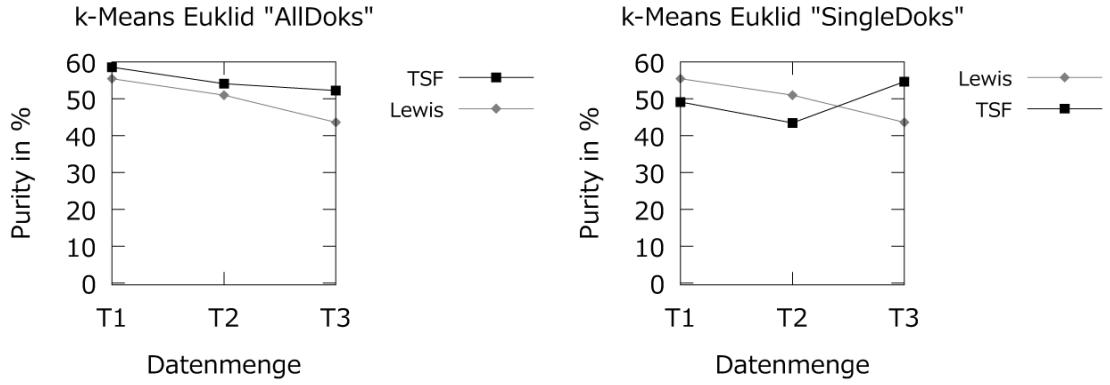


Abbildung 5.9: Ergebnis der Evaluation des durch den k-Means-Clusterer mit Euklid erzeugten Clusterings für die Purity für das Experiment Clus1RV

Betrachtet man dagegen die TSF-Feature-Erzeugung „SingleDoks“, so erreichen die einzelwortbasierten Feature-Vektoren in 2 von 3 Fällen ein besseres Ergebnis für die Purity. So unterscheiden sich die erreichten Ergebnisse im Durchschnitt über die drei Datenmengen um gerundet 8,3 Prozentpunkte voneinander. Dabei wird der größte Abstand von 11,05 Prozentpunkten bei Datenmenge T3 erreicht und der kleinste von 6,35 bei Datenmenge T1.

Zusammengefasst ergibt sich für den k-Means-Clusterer mit euklidischem Distanzmaß für die Purity folgendes Ergebnis:

* Bei „AllDoks“ erreichen die TSF-Feature-Vektoren ein besseres Ergebnis als die einzelwortbasierten Feature-Vektoren.
* Bei „SingleDoks“ erreichen die einzelwortbasierten Feature-Vektoren für zwei Datenmengen bessere Ergebnisse als die TSF-Feature-Vektoren.
* Insgesamt ergibt sich damit ein Gleichstand, da bei der ersten Art der TSF-Featurerzeugung die TSF-Feature-Vektoren ein besseres Ergebnis erreichen und bei der zweiten Art die einzelwortbasierten Feature-Vektoren.
* Nach diesem ersten Teilexperiment scheint die TSF-Featurerzeugung, bei der alle Dokumente zur Feature-Erzeugung herangezogen werden, besser zu sein als die Art, bei der jedes Dokument einzeln betrachtet wird.
* Berücksichtigt man nicht nur die besten Ergebnisse pro Datenmenge, sondern die gesamte Tabelle, so fällt auf, dass sich die Ergebnisse über alle Durchläufe teilweise gravierend unterscheiden. Das gilt nicht nur für die TSF-Feature-Vektoren, sondern auch für die einzelwortbasierten Vektoren. Diese sehr unterschiedlichen Ergebnisse resultieren wahrscheinlich aus der Güte der ersten zufällig gewählten Clustermittelpunkte. Sind diese „schlecht“ gewählt, so kann insgesamt kein „gutes“ Ergebnis erreicht werden und umgekehrt.

– Ergebnisvergleich k-Means mit Cosinus
  Bildet man die Ergebnisse, die sich durch die Auswertung des k-Means-Clusterers mit Cosinus als Distanzmaß für die Purity für alle Datenmengen ergeben, in Diagrammen ab, so erhält man die Abbildung 5.10 auf S. 487.

  Im Fall der TSF-Feature-Erzeugung „AllDoks“ ergibt sich über die drei Datenmengen in 2 von 3 Fällen für den k-Means-Clusterer mit Cosinus als Distanzmaß ein besseres Ergebnis für die TSF-Feature-Vektoren als für die einzelwortbasierten Feature-Vektoren. Im Durchschnitt unterscheidet sich die Purity für die beiden Verfahren um gerundet 3,78 Prozentpunkte, wobei der geringste Abstand 2,7 Prozentpunkte und der maximale Abstand 4,35 Prozentpunkte beträgt. Ersterer wird für Datenmenge T3 erreicht, also für ein besseres Ergebnis der TSF-Feature-Vektoren, und letzterer für Datenmenge T2, also für ein besseres Ergebnis der einzelwortbasierten Feature-Vektoren.

Bei der TSF-Feature-Erzeugung „SingleDoks“ erhält man ein sehr ähnliches Bild. Auch in diesem Fall erreichen die TSF-Feature-Vektoren in 2 von 3 Fällen ein besseres Ergebnis für die Purity. Die erreichten Ergebnisse unterscheiden sich im Durchschnitt über die drei Datenmengen um gerundet 3,28 Prozentpunkte voneinander. Dabei wird der größte Abstand von 4,6 Prozentpunkten bei Datenmenge T3 erreicht und der kleinste Abstand von 1,8 bei Datenmenge T2.

Zusammengefasst ergibt sich für den k-Means-Clusterer mit Cosinus als Distanzmaß für die Purity folgendes Ergebnis:

* Bei „AllDoks“ erreichen die TSF-Feature-Vektoren für 2 von 3 Datenmengen ein besseres Ergebnis als die einzelwortbasierten Feature-Vektoren. Insgesamt wird dies als ein besseres Ergebnis für die TSF-Feature-Vektoren gewertet.
* Bei „SingleDoks“ ist das Ergebnis genauso.
* Insgesamt ergibt sich damit, dass für dieses Teilexperiment des Experiments 7 die TSF-Feature-Vektoren bessere Ergebnisse erreichen als die einzelwortbasierten Feature-Vektoren.
* Nach diesem zweiten Teilexperiment des Experiments 7 scheint die TSF-Featurerzeugung, bei der jedes Dokument einzeln zur Feature-Erzeugung herangezogen wird, besser zu sein als die Art, bei der alle Dokumente betrachtet werden.
* Berücksichtigt man nicht nur die besten Ergebnisse pro Datenmenge, sondern die gesamte Tabelle, so fällt auf, dass sich die Ergebnisse über alle Durchläufe teilweise gravierend unterscheiden. Das gilt nicht nur für die TSF-Feature-Vektoren sondern auch für die einzelwortbasierten Feature-Vektoren. Das resultiert wahrscheinlich ebenfalls aus der Auswahl der ersten Clustermittelpunkte.

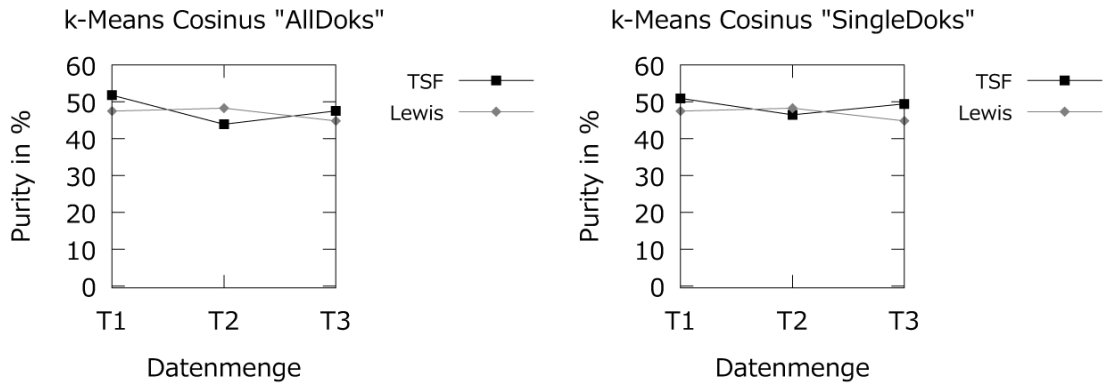


Abbildung 5.10: Ergebnis der Evaluation des durch den k-Means-Clusterer mit Cosinus erzeugten Clusterings für die Purity für das Experiment Clus1RV

– Ergebnisvergleich GAAC
  Bildet man die Ergebnisse, die sich durch die Auswertung des GAAC-Clusterers für die Purity für alle Datenteilmengen ergeben, in Diagrammen ab, so erhält man Folgendes:

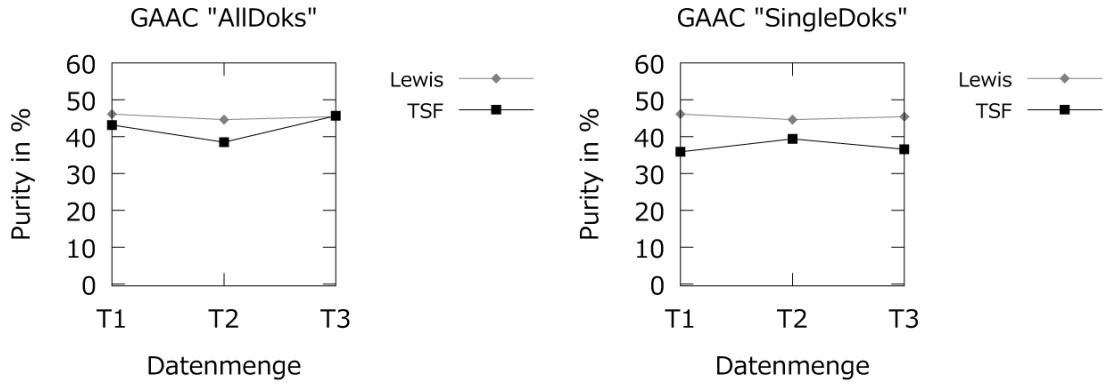


Abbildung 5.11: Ergebnis der Evaluation des durch den GAAC-Clusterer erzeugten Clusterings für die Purity für das Experiment Clus1RV

Im Fall der TSF-Feature-Erzeugung „AllDoks“ ergibt sich in zwei von drei Fällen für den GAAC-Clusterer ein besseres Ergebnis für die einzelwortbasierten Feature-Vektoren als für die TSF-Feature-Vektoren. In einem Fall

erreichen die TSF-Feature-Vektoren ein leicht besseres Ergebnis. Im Durchschnitt unterscheiden sich die Ergebnisse beider Verfahren um gerundet 3,1 Prozentpunkte, wobei der geringste Abstand 0,25 Prozentpunkte und der maximale Abstand 6,1 Prozentpunkte beträgt.

Betrachtet man dagegen die TSF-Feature-Erzeugung „SingleDoks“, so erreichen die einzelwortbasierten Feature-Vektoren in allen Fällen ein besseres Ergebnis. So liegen die Ergebnisse für die Purity für die einzelwortbasierten Feature-Vektoren im Durchschnitt 8,1 Prozentpunkte über den Ergebnissen für die TSF-Feature-Vektoren. Der kleinste Abstand beträgt dabei 5,2 Prozentpunkte und der größte Abstand 10,25 Prozentpunkte.

Zusammengefasst ergibt sich für den GAAC-Clusterer für die Purity folgendes Ergebnis:

* Bei „AllDoks“ erreichen die einzelwortbasierten Feature-Vektoren ein besseres Ergebnis als die TSF-Feature-Vektoren. Der fast Gleichstand bei Datenmenge T3 fällt dabei nicht ins Gewicht.
* Bei „SingleDoks“ erreichen die einzelwortbasierten Feature-Vektoren über alle Datenmengen ein deutlich besseres Ergebnis als die TSF-Feature-Vektoren.
* Insgesamt ergibt sich damit, dass für dieses Teilexperiment des Experiments 7 die einzelwortbasierten Feature-Vektoren bessere Ergebnisse erreichen als die TSF-Feature-Vektoren.
* Nach diesem dritten Teilexperiment des Experiments 7 scheint die TSF-Featurerzeugung, bei der alle Dokumente zur Feature-Erzeugung herangezogen werden, besser zu sein als die Art, bei der jedes Dokument einzeln betrachtet wird.

– Ergebnisvergleich über alle Algorithmen und Distanzmaße
Vergleicht man die Ergebnisse des einzelwortbasierten Verfahrens und des Verfahrens mit TSF-Features über alle Algorithmen miteinander, so stellt man Folgendes fest: Es herrscht ein Gleichstand zwischen den Verfahren, wenn die Ergebnisse mit dem Qualitätskriterium Purity evaluiert werden.

Dieses erste Experiment für das Clustern kann also die Vermutung, dass wortübergreifende Features qualitativ bessere Ergebnisse beim Clustern von natürlichsprachlichen Dokumenten liefern als Verfahren, die natürlichsprachliche Dokumente in Einzelworte aufteilen und diese als Features für das Clustern benutzen, weder stützen noch widerlegen.

Als Tabelle zusammengefasst ergibt sich für das erste Experiment folgendes Bild[1]:

Tabelle 5.8: Ergebnisse des Experiments Clus1RV

Verfahren	Algorithmus		
	k-Means Euklid	k-Means Cosinus	GAAC
„AllDoks“-Feature-Vektoren	✓	✓	
„SingleDoks“-Feature-Vektoren		✓	
Lewis-Feature-Vektoren	✓		✓✓

Damit erreichen die TSF-Feature-Vektoren, wenn man beide Erzeugungsarten zusammenfasst, in 3 von 6 Fällen ein besseres Ergebnis als die einzelwortbasierten Feature-Vektoren. Gleiches gilt umgekehrt ebenfalls. Betrachtet man die beiden Erzeugungsarten der TSF-Features einzeln, so ergibt sich in 2 von 3 Fällen ein besseres Ergebnis der TSF-Feature-Vektoren bei der Erzeugungsart „AllDoks“. Dagegen erreichen die TSF-Feature-Vektoren für die Erzeugungsart „SingleDoks“ nur in 1 von 3 Fällen ein besseres Ergebnis als die einzelwortbasierten Feature-Vektoren. Insgesamt weist dieses erste Cluster-Experiment damit darauf hin, dass beim Clustern mit TSF-Feature-Vektoren bessere Ergebnisse erzielt werden, wenn Features aus allen zu clusternden Dokumenten erzeugt werden und nicht nur für jedes Dokument einzeln.

5.3.3.2 Experiment 8

- ID
  Clus2RV

- Bezeichnung
  vektorbasiertes Region-Clustern von Daten der Reuters-Daten RCV1-v2 mit Rand Index als Evaluationsmaß

1 Ein Häkchen zeigt das Verfahren an, das für den betreffenden Algorithmus die höchste Qualität über alle Datenmengen erreicht hat. Zwei Häkchen in der gleichen Spalte für Lewis bedeuten, dass die einzelwortbasierten Feature-Vektoren gegenüber beiden Arten der TSF-Feature-Erzeugung besser abgeschnitten haben.

- Daten
  Es werden die gleichen Daten verwendet wie in Experiment Clus1RV.
- Ähnlichkeit
  vektorbasiert
- Algorithmen
  Es werden die gleichen Algorithmen verwendet wie in Experiment Clus1RV.
- Anzahl Cluster
  4
- Evaluation
  Rand Index
- Ergebnisse[1]
  Die Ergebnisse für den k-Means-Clusterer mit euklidischem Distanzmaß befinden sich in den Tabellen 5.9 bis 5.11 auf den Seiten 490 bis 491.
  Die Ergebnisse für den k-Means-Clusterer mit Cosinus-Distanzmaß befinden sich in den Tabellen 5.12 bis 5.14 auf den Seiten 491 bis 492.
  Die Ergebnisse für den GAAC-Clusterer befinden sich in Tabelle 5.15 auf S. 493.

Tabelle 5.9: Ergebnisse des Experiments Clus2RV für Datenmenge T1 und dem k-Means-Clusterer mit Euklid-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	Rand Index in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	68,52
	Region	1	„SingleDoks“	64,06
Lewis-Feature-Vektoren	Region	1	-	65,06
TSF-Feature-Vektoren	Region	2	„AllDoks“	51,03
	Region	2	„SingleDoks“	57,74
Lewis-Feature-Vektoren	Region	2	-	66,45
TSF-Feature-Vektoren	Region	3	„AllDoks“	52,88
	Region	3	„SingleDoks“	58,45
Lewis-Feature-Vektoren	Region	3	-	62,80

1 In allen Tabellen werden die Ergebnisse auf zwei Stellen nach dem Komma gerundet. Das jeweils dunkel hinterlegte Tabellenfeld kennzeichnet beim k-Means-Clusterer das beste Ergebnis über die drei Durchläufe pro Feature-Erzeugungsmethode.

Tabelle 5.10: Ergebnisse des Experiments Clus2RV für Datenmenge T2 und dem k-Means-Clusterer mit Euklid-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	Rand Index in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	62,30
	Region	1	„SingleDoks“	39,23
Lewis-Feature-Vektoren	Region	1	-	64,60
TSF-Feature-Vektoren	Region	2	„AllDoks“	56,36
	Region	2	„SingleDoks“	57,53
Lewis-Feature-Vektoren	Region	2	-	62,53
TSF-Feature-Vektoren	Region	3	„AllDoks“	56,38
	Region	3	„SingleDoks“	63,69
Lewis-Feature-Vektoren	Region	3	-	62,88

Tabelle 5.11: Ergebnisse des Experiments Clus2RV für Datenmenge T3 und dem k-Means-Clusterer mit Euklid-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	Rand Index in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	54,09
	Region	1	„SingleDoks“	59,67
Lewis-Feature-Vektoren	Region	1	-	52,05
TSF-Feature-Vektoren	Region	2	„AllDoks“	63,21
	Region	2	„SingleDoks“	67,36
Lewis-Feature-Vektoren	Region	2	-	56,68
TSF-Feature-Vektoren	Region	3	„AllDoks“	55,51
	Region	3	„SingleDoks“	52,94
Lewis-Feature-Vektoren	Region	3	-	59,25

Tabelle 5.12: Ergebnisse des Experiments Clus2RV für Datenmenge T1 und dem k-Means-Clusterer mit Cosinus-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	Rand Index in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	62,01
	Region	1	„SingleDoks“	57,58
Lewis-Feature-Vektoren	Region	1	-	64,89

wird auf der nächsten Seite fortgesetzt

Tabelle 5.12: Ergebnisse des Experiments Clus2RV für Datenmenge T1 und dem k-Means-Clusterer mit Cosinus-Distanzmaß (Fortsetzung)

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	Rand Index in %
TSF-Feature-Vektoren	Region	2	„AllDoks“	49,96
	Region	2	„SingleDoks“	62,96
Lewis-Feature-Vektoren	Region	2	-	60,35
TSF-Feature-Vektoren	Region	3	„AllDoks“	59,14
	Region	3	„SingleDoks“	62,86
Lewis-Feature-Vektoren	Region	3	-	66,40

Tabelle 5.13: Ergebnisse des Experiments Clus2RV für Datenmenge T2 und dem k-Means-Clusterer mit Cosinus-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	Rand Index in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	50,20
	Region	1	„SingleDoks“	61,45
Lewis-Feature-Vektoren	Region	1	-	62,86
TSF-Feature-Vektoren	Region	2	„AllDoks“	50,09
	Region	2	„SingleDoks“	60,18
Lewis-Feature-Vektoren	Region	2	-	64,43
TSF-Feature-Vektoren	Region	3	„AllDoks“	48,35
	Region	3	„SingleDoks“	61,20
Lewis-Feature-Vektoren	Region	3	-	62,93

Tabelle 5.14: Ergebnisse des Experiments Clus2RV für Datenmenge T3 und dem k-Means-Clusterer mit Cosinus-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	Rand Index in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	45,63
	Region	1	„SingleDoks“	56,81
Lewis-Feature-Vektoren	Region	1	-	59,91
TSF-Feature-Vektoren	Region	2	„AllDoks“	54,43
	Region	2	„SingleDoks“	59,85
Lewis-Feature-Vektoren	Region	2	-	62,07

wird auf der nächsten Seite fortgesetzt

Tabelle 5.14: Ergebnisse des Experiments Clus2RV für Datenmenge T3 und dem k-Means-Clusterer mit Cosinus-Distanzmaß (Fortsetzung)

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	Rand Index in %
TSF-Feature-Vektoren	Region	3	„AllDoks“	49,33
	Region	3	„SingleDoks“	59,04
Lewis-Feature-Vektoren	Region	3	-	60,21

Tabelle 5.15: Ergebnisse des Experiments Clus2RV für Datenmenge T1 bis T3 und dem GAAC-Clusterer

Verfahren	Daten-menge	Klassen-familie	Methode TSF-Features	Rand Index in %
TSF-Feature-Vektoren	1	Region	„AllDoks“	61,94
	1	Region	„SingleDoks“	51,69
Lewis-Feature-Vektoren	1	Region	-	64,69
TSF-Feature-Vektoren	2	Region	„AllDoks“	56,90
	2	Region	„SingleDoks“	53,93
Lewis-Feature-Vektoren	2	Region	-	64,35
TSF-Feature-Vektoren	3	Region	„AllDoks“	60,43
	3	Region	„SingleDoks“	53,12
Lewis-Feature-Vektoren	3	Region	-	64,20

- Interpretation der Ergebnisse
  - Einführung

    Im Folgenden werden die vorgestellten Ergebnisse für die auf den Suffix Arrays basierenden Vektoren und für die Lewis-Feature-Vektoren miteinander verglichen. Dabei werden zunächst die besten Ergebnisse pro Algorithmus und Distanzmaß über die drei Datenmengen miteinander verglichen und abschließend erfolgt ein Vergleich für das gesamte Experiment.

– Ergebnisvergleich k-Means mit Euklid
  Bildet man die Ergebnisse, die sich durch die Auswertung des k-Means-Clusterers mit Euklid als Distanzmaß für den Rand Index für alle Datenmengen ergeben, in Diagrammen ab, so erhält man Folgendes:

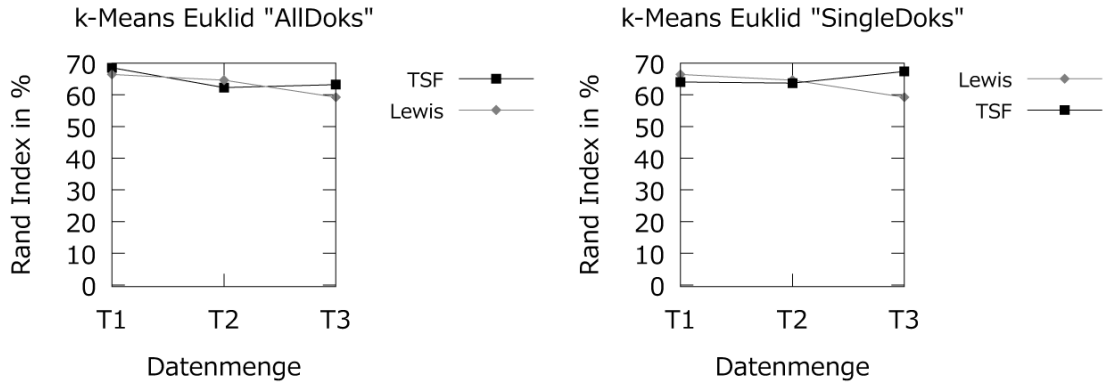


Abbildung 5.12: Ergebnis der Evaluation des durch den k-Means-Clusterer mit Euklid erzeugten Clusterings für den Rand Index für das Experiment Clus2RV

Im Fall der TSF-Feature-Erzeugung „AllDoks“ ergibt sich über die drei Datenmengen für den k-Means-Clusterer mit euklidischem Distanzmaß, dass in 2 von 3 Fällen die TSF-Feature-Vektoren bessere Ergebnisse erzielen als die einzelwortbasierten Feature-Vektoren. Im Durchschnitt unterscheiden sich die Ergebnisse um gerundet 2,78 Prozentpunkte, wobei der minimale Abstand 2,07 Prozentpunkte und der maximale Abstand 3,97 Prozentpunkte beträgt.

Betrachtet man dagegen die TSF-Feature-Erzeugung „SingleDoks“, so erreichen die einzelwortbasierten Feature-Vektoren in 2 von 3 Fällen ein besseres Ergebnis für den Rand Index. So unterscheiden sich die erreichten Ergebnisse im Durchschnitt über die drei Datenmengen um gerundet 3,8 Prozentpunkte voneinander. Dabei wird der größte Abstand von 8,11 Prozentpunkten bei Datenmenge T3 erreicht, also in dem Fall, dass die TSF-Feature-Vektoren ein besseres Ergebnis erreichen, und der kleinste Abstand von 0,91 Prozentpunkten bei Datenmenge T2.

Zusammengefasst ergibt sich für den k-Means-Clusterer mit euklidischem Distanzmaß für den Rand Index folgendes Ergebnis:

* Bei „AllDoks“ erreichen die TSF-Feature-Vektoren in 2 von 3 Fällen ein besseres Ergebnis als die einzelwortbasierten Feature-Vektoren.
* Bei „SingleDoks“ erreichen die einzelwortbasierten Feature-Vektoren für zwei Datenmengen bessere Ergebnisse als die TSF-Feature-Vektoren.
* Insgesamt ergibt sich damit ein Gleichstand, da bei der ersten Art der TSF-Feature-Erzeugung die TSF-Feature-Vektoren ein besseres Ergebnis erreichen und bei der zweiten Art die einzelwortbasierten Feature-Vektoren.
* Nach diesem ersten Teilexperiment von Experiment 8 scheint die TSF-Feature-Erzeugung, bei der alle Dokumente zur Feature-Erzeugung herangezogen werden, besser zu sein als die Art, bei der jedes Dokument einzeln betrachtet wird.
* Auch bei diesem Teilexperiment scheinen die Ergebnisse von einer „guten“ Wahl der ersten Clustermittelpunkte abhängig zu sein.

– Ergebnisvergleich k-Means mit Cosinus
Bildet man die Ergebnisse, die sich durch die Auswertung des k-Means-Clusterers mit Cosinus als Distanzmaß für den Rand Index für alle Datenmengen ergeben, in Diagrammen ab, so erhält man die Abbildung 5.13 auf S. 496.

Im Fall der TSF-Feature-Erzeugung „AllDoks“ ergibt sich über die drei Datenmengen für den k-Means-Clusterer mit Cosinus als Distanzmaß ein besseres Ergebnis für die einzelwortbasierten Feature-Vektoren als für die TSF-Feature-Vektoren. Im Durchschnitt unterscheidet sich der Rand Index für die beiden Verfahren um gerundet 8,75 Prozentpunkte, wobei der geringste Abstand 4,38 Prozentpunkte und der maximale Abstand 14,23 Prozentpunkte beträgt.

Bei der TSF-Feature-Erzeugung „SingleDoks“ erhält man ein sehr ähnliches Bild. Auch in diesem Fall erreichen die einzelwortbasierten Feature-Vektoren ein besseres Ergebnis beim Rand Index, jedoch weitaus weniger deutlich. Die erreichten Ergebnisse im Durchschnitt über die drei Datenmengen unterscheiden sich um gerundet 2,88 Prozentpunkte voneinander. Dabei beträgt der größte Abstand 3,44 Prozentpunkte und der kleinste Abstand 2,22 Prozentpunkte.

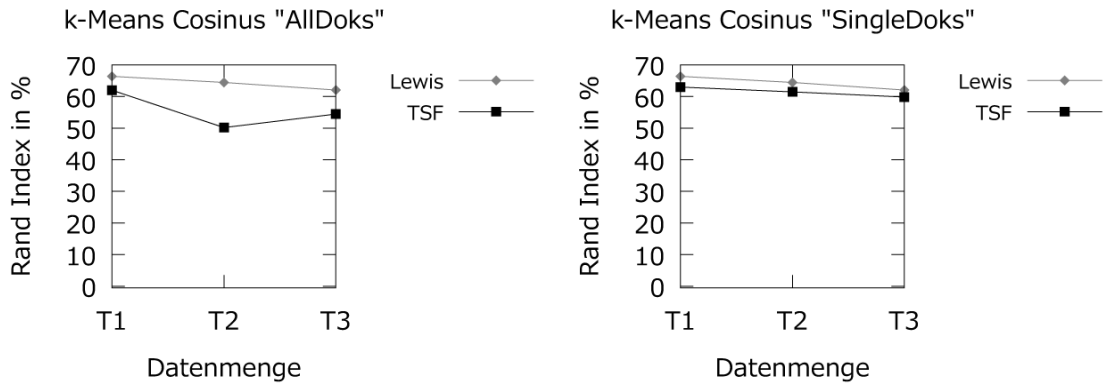


Abbildung 5.13: Ergebnis der Evaluation des durch den k-Means-Clusterer mit Cosinus erzeugten Clusterings für den Rand Index für das Experiment Clus2RV

Zusammengefasst ergibt sich für den k-Means-Clusterer mit Cosinus als Distanzmaß für den Rand Index folgendes Ergebnis:

* Sowohl bei „AllDoks“ als auch bei „SingleDoks“ erreichen die einzelwortbasierten Feature-Vektoren bessere Ergebnisse als die TSF-Feature-Vektoren.
* Nach diesem zweiten Teilexperiment des Experiments 8 scheint die TSF-Featurerzeugung, bei der jedes Dokument einzeln zur Feature-Erzeugung herangezogen wird, besser zu sein als die Art, bei der alle Dokumente betrachtet werden.
* Auch in diesem Teilexperiment wird deutlich, wie sehr die Ergebnisse von der ersten Wahl der Clustermittelpunkte beeinflusst zu sein scheinen.

– Ergebnisvergleich GAAC
Bildet man die Ergebnisse, die sich durch die Auswertung des GAAC-Clusterers für den Rand Index für alle Datenmengen ergeben, in Diagrammen ab, so erhält man die Abbildung 5.14 auf dieser Seite.

Im Fall der TSF-Feature-Erzeugung „AllDoks“ ergibt sich in allen Fällen für den GAAC-Clusterer ein besseres Ergebnis für die einzelwortbasierten Feature-Vektoren als für die TSF-Feature-Vektoren. Im Durchschnitt unterscheiden sich die Ergebnisse beider Verfahren um gerundet 4,65 Prozentpunkte, wobei der geringste Abstand 2,75 Prozentpunkte und der maximale Abstand 7,44 Prozentpunkte beträgt.

Betrachtet man die TSF-Feature-Erzeugung „SingleDoks“, so erreichen die einzelwortbasierten Feature-Vektoren in allen Fällen ebenfalls ein besseres Ergebnis. So liegen die Ergebnisse für den Rand Index für die einzelwortbasierten Feature-Vektoren im Schnitt 11,5 Prozentpunkte über den Ergebnissen für die TSF-Feature-Vektoren. Der kleinste Abstand beträgt dabei 10,42 Prozentpunkte und der größte Abstand 13,01 Prozentpunkte.

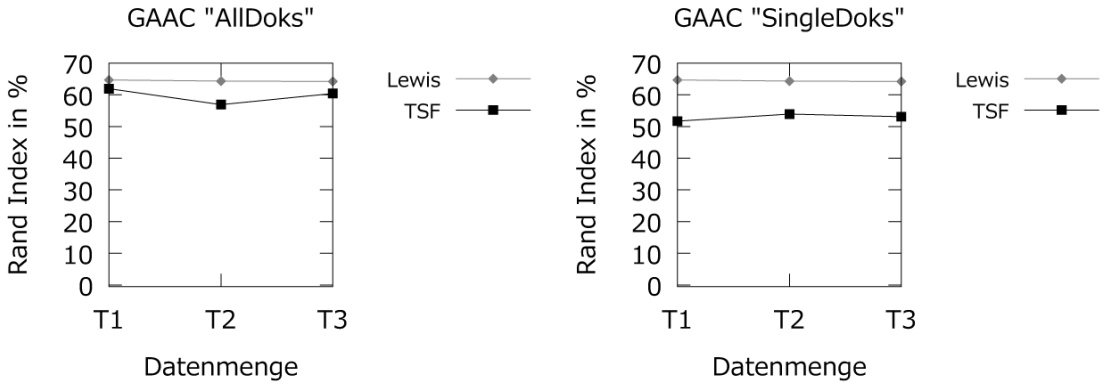


Abbildung 5.14: Ergebnis der Evaluation des durch den GAAC-Clusterer erzeugten Clusterings für den Rand Index für das Experiment Clus2RV

Zusammengefasst ergibt sich für den GAAC-Clusterer für den Rand Index folgendes Ergebnis:

* Bei beiden Arten der TSF-Feature-Erzeugung erreichen die einzelwortbasierten Feature-Vektoren bessere Ergebnisse als die TSF-Feature-Vektoren.
* Nach diesem dritten Teilexperiment des Experiments 8 scheint die TSF-Featurerzeugung, bei der alle Dokumente zur Feature-Erzeugung herangezogen werden, besser zu sein als die Art, bei der jedes Dokument einzeln betrachtet wird.

– Ergebnisvergleich über alle Algorithmen und Distanzmaße
Vergleicht man die Ergebnisse des einzelwortbasierten Verfahrens und des Verfahrens mit TSF-Features über alle Algorithmen miteinander, so stellt man Folgendes fest: Die einzelwortbasierten Feature-Vektoren erreichen für den Rand Index in 5 von 6 Fällen ein besseres Ergebnis als die TSF-Feature-Vektoren.

Dieses zweite Cluster-Experiment kann also die Vermutung, dass wortübergreifende Features qualitativ bessere Ergebnisse beim Clustern von natürlichsprachlichen Dokumenten liefern als Verfahren, die natürlichsprachliche Dokumente in Einzelworte aufteilen und diese als Features für das Clustern benutzen, „mit 5:1“ widerlegen.

Als Tabelle zusammengefasst ergibt sich für das zweite Cluster-Experiment folgendes Bild[1]:

Tabelle 5.16: Ergebnisse des Experiments Clus2RV

Verfahren	Algorithmus		
	k-Means Euklid	k-Means Cosinus	GAAC
„AllDoks“-Feature-Vektoren	✓		
„SingleDoks“-Feature-Vektoren			
Lewis-Feature-Vektoren	✓	✓✓	✓✓

Damit erreichen die TSF-Feature-Vektoren, wenn man beide Erzeugungsarten zusammenfasst, in nur 1 von 6 Fällen ein besseres Ergebnis als die einzelwortbasierten Feature-Vektoren. Betrachtet man die beiden Erzeugungsarten der TSF-Features einzeln, so ergibt sich in 1 von 3 Fällen ein besseres Ergebnis der TSF-Feature-Vektoren bei der Erzeugungsart „AllDoks“. Dagegen erreichen die TSF-Feature-Vektoren für die Erzeugungsart „SingleDoks“ in gar keinem Fall ein besseres Ergebnis als die einzelwortbasierten Feature-Vektoren. Insgesamt weist dieses zweite Cluster-Experiment damit darauf hin, dass beim Clustern mit TSF-Feature-Vektoren bessere Ergebnisse erzielt werden, wenn Features aus allen zu clusternden Dokumenten erzeugt werden und nicht nur für jedes Dokument einzeln.

5.3.3.3 Experiment 9

- ID

  Clus3RV

1 Ein Häkchen zeigt das Verfahren an, das für den betreffenden Algorithmus die höchste Qualität über alle Datenmengen erreicht hat. Zwei Häkchen in der gleichen Spalte für Lewis bedeuten, dass die einzelwortbasierten Feature-Vektoren gegenüber beiden Arten der TSF-Feature-Erzeugung besser abgeschnitten haben.

- Bezeichnung
  vektorbasiertes Region-Clustern von Daten der Reuters-Daten RCV1-v2 mit F_1 als Evaluationsmaß

- Daten
  Es werden die gleichen Daten verwendet wie in Experiment Clus1RV.

- Ähnlichkeit
  vektorbasiert

- Algorithmen
  Es werden die gleichen Algorithmen verwendet wie in Experiment Clus1RV.

- Anzahl Cluster
  4

- Evaluation
  F_1-Maß

- Ergebnisse[1]
  Die Ergebnisse für den k-Means-Clusterer mit euklidischem Distanzmaß befinden sich in den Tabellen 5.17 bis 5.19 auf den Seiten 499 bis 500.
  Die Ergebnisse für den k-Means-Clusterer mit Cosinus-Distanzmaß befinden sich in den Tabellen 5.20 bis 5.22 auf den Seiten 501 bis 501.
  Die Ergebnisse für den GAAC-Clusterer befinden sich in Tabelle 5.23 auf S. 502.

Tabelle 5.17: Ergebnisse des Experiments Clus3RV für Datenmenge T1 und dem k-Means-Clusterer mit Euklid-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	F_1 in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	44,28
	Region	1	„SingleDoks“	35,27
Lewis-Feature-Vektoren	Region	1	-	39,88
TSF-Feature-Vektoren	Region	2	„AllDoks“	37,53
	Region	2	„SingleDoks“	35,97
Lewis-Feature-Vektoren	Region	2	-	38,72

wird auf der nächsten Seite fortgesetzt

1 In allen Tabellen werden die Ergebnisse auf zwei Stellen nach dem Komma gerundet. Das jeweils dunkel hinterlegte Tabellenfeld kennzeichnet beim k-Means-Clusterer das beste Ergebnis über die drei Durchläufe pro Feature-Erzeugungsmethode.

Tabelle 5.17: Ergebnisse des Experiments Clus3RV für Datenmenge T1 und dem k-Means-Clusterer mit Euklid-Distanzmaß (Fortsetzung)

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	F_1 in %
TSF-Feature-Vektoren	Region	3	„AllDoks“	36,21
	Region	3	„SingleDoks“	35,58
Lewis-Feature-Vektoren	Region	3	-	35,99

Tabelle 5.18: Ergebnisse des Experiments Clus3RV für Datenmenge T2 und dem k-Means-Clusterer mit Euklid-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	F_1 in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	36,00
	Region	1	„SingleDoks“	40,06
Lewis-Feature-Vektoren	Region	1	-	36,48
TSF-Feature-Vektoren	Region	2	„AllDoks“	37,02
	Region	2	„SingleDoks“	37,02
Lewis-Feature-Vektoren	Region	2	-	36,23
TSF-Feature-Vektoren	Region	3	„AllDoks“	42,08
	Region	3	„SingleDoks“	34,19
Lewis-Feature-Vektoren	Region	3	-	36,19

Tabelle 5.19: Ergebnisse des Experiments Clus3RV für Datenmenge T3 und dem k-Means-Clusterer mit Euklid-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	F_1 in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	36,72
	Region	1	„SingleDoks“	34,03
Lewis-Feature-Vektoren	Region	1	-	38,51
TSF-Feature-Vektoren	Region	2	„AllDoks“	37,41
	Region	2	„SingleDoks“	45,38
Lewis-Feature-Vektoren	Region	2	-	38,28
TSF-Feature-Vektoren	Region	3	„AllDoks“	41,14
	Region	3	„SingleDoks“	35,59
Lewis-Feature-Vektoren	Region	3	-	36,06

Tabelle 5.20: Ergebnisse des Experiments Clus3RV für Datenmenge T1 und dem k-Means-Clusterer mit Cosinus-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	F_1 in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	40,14
	Region	1	„SingleDoks“	34,69
Lewis-Feature-Vektoren	Region	1	-	34,92
TSF-Feature-Vektoren	Region	2	„AllDoks“	36,24
	Region	2	„SingleDoks“	37,55
Lewis-Feature-Vektoren	Region	2	-	36,14
TSF-Feature-Vektoren	Region	3	„AllDoks“	39,92
	Region	3	„SingleDoks“	35,90
Lewis-Feature-Vektoren	Region	3	-	35,83

Tabelle 5.21: Ergebnisse des Experiments Clus3RV für Datenmenge T2 und dem k-Means-Clusterer mit Cosinus-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	F_1 in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	40,58
	Region	1	„SingleDoks“	35,56
Lewis-Feature-Vektoren	Region	1	-	36,09
TSF-Feature-Vektoren	Region	2	„AllDoks“	41,20
	Region	2	„SingleDoks“	36,13
Lewis-Feature-Vektoren	Region	2	-	35,30
TSF-Feature-Vektoren	Region	3	„AllDoks“	39,32
	Region	3	„SingleDoks“	36,52
Lewis-Feature-Vektoren	Region	3	-	36,83

Tabelle 5.22: Ergebnisse des Experiments Clus3RV für Datenmenge T3 und dem k-Means-Clusterer mit Cosinus-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	F_1 in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	38,30
	Region	1	„SingleDoks“	32,99
Lewis-Feature-Vektoren	Region	1	-	36,58

wird auf der nächsten Seite fortgesetzt

Tabelle 5.22: Ergebnisse des Experiments Clus3RV für Datenmenge T3 und dem k-Means-Clusterer mit Cosinus-Distanzmaß (Fortsetzung)

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	F_1 in %
TSF-Feature-Vektoren	Region	2	„AllDoks“	41,62
	Region	2	„SingleDoks“	34,43
Lewis-Feature-Vektoren	Region	2	-	33,21
TSF-Feature-Vektoren	Region	3	„AllDoks“	38,76
	Region	3	„SingleDoks“	38,83
Lewis-Feature-Vektoren	Region	3	-	36,33

Tabelle 5.23: Ergebnisse des Experiments Clus3RV für Datenmenge T1 bis T3 und dem GAAC-Clusterer

Verfahren	Daten-menge	Klassen-familie	Methode TSF-Features	F_1 in %
TSF-Feature-Vektoren	1	Region	„AllDoks“	33,61
	1	Region	„SingleDoks“	34,87
Lewis-Feature-Vektoren	1	Region	-	36,81
TSF-Feature-Vektoren	2	Region	„AllDoks“	33,51
	2	Region	„SingleDoks“	34,88
Lewis-Feature-Vektoren	2	Region	-	34,91
TSF-Feature-Vektoren	3	Region	„AllDoks“	35,98
	3	Region	„SingleDoks“	35,50
Lewis-Feature-Vektoren	3	Region	-	36,84

- Interpretation der Ergebnisse
  - Einführung

    Im Folgenden werden die vorgestellten Ergebnisse für die auf den Suffix Arrays basierenden Vektoren und für die Lewis-Feature-Vektoren miteinander verglichen. Dabei werden zunächst die besten Ergebnisse pro Algorithmus und Distanzmaß über die drei Datenmengen miteinander verglichen und abschließend erfolgt ein Vergleich für das gesamte Experiment.

– Ergebnisvergleich k-Means mit Euklid
  Bildet man die Ergebnisse, die sich durch die Auswertung des k-Means-Clusterers mit Euklid als Distanzmaß für das F_1-Maß für alle Datenmengen ergeben, in Diagrammen ab, so erhält man Folgendes:

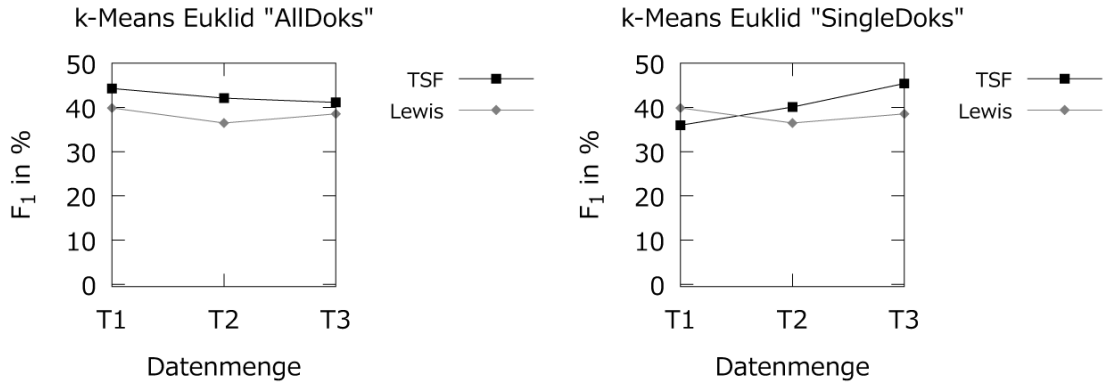


Abbildung 5.15: Ergebnis der Evaluation des durch den k-Means-Clusterer mit Euklid erzeugten Clusterings für das F_1-Maß für das Experiment Clus3RV

Im Fall der TSF-Feature-Erzeugung „AllDoks“ ergibt sich über die drei Datenmengen für den k-Means-Clusterer mit euklidischem Distanzmaß, dass die TSF-Feature-Vektoren bessere Ergebnisse erzielen als die einzelwortbasierten Feature-Vektoren. Im Durchschnitt unterscheiden sich die Ergebnisse um rund 4,21 Prozentpunkte, wobei der minimale Abstand 2,63 Prozentpunkte und der maximale Abstand 5,6 Prozentpunkte beträgt.

Betrachtet man dagegen die TSF-Feature-Erzeugung „SingleDoks“, so erreichen die TSF-Feature-Vektoren in 2 von 3 Fällen ein besseres Ergebnis beim F_1-Maß. So unterscheiden sich die erreichten Ergebnisse im Durchschnitt über die drei Datenmengen um rund 4,79 Prozentpunkte voneinander. Dabei wird der größte Abstand von 6,87 Prozentpunkten bei Datenmenge T3 erreicht, also in dem Fall, dass die TSF-Feature-Vektoren ein besseres Ergebnis erreichen, und der kleinste von 3,58 bei Datenmenge T2 mit ebenfalls einem besseren Ergebnis für die TSF-Feature-Vektoren.

Zusammengefasst ergibt sich für den k-Means-Clusterer mit euklidischem Distanzmaß für das F_1-Maß folgendes Ergebnis:

* Bei „AllDoks“ erreichen die TSF-Feature-Vektoren in allen Fällen ein besseres Ergebnis als die einzelwortbasierten Feature-Vektoren.

* Bei „SingleDoks“ erreichen die TSF-Feature-Vektoren für zwei Datenmengen bessere Ergebnisse als die einzelwortbasierten Feature-Vektoren.

* Insgesamt ergibt sich damit, dass in diesem ersten Teilexperiment von Experiment 9 die TSF-Feature-Vektoren ein besseres Ergebnis erreichen, als die einzelwortbasierten Feature-Vektoren.

* Nach diesem ersten Teilexperiment von Experiment 9 scheint die TSF-Featurerzeugung, bei der alle Dokumente zur Feature-Erzeugung herangezogen werden, besser zu sein als die Art, bei der jedes Dokument einzeln betrachtet wird.

* Auch bei diesem Teilexperiment scheinen die Ergebnisse von einer „guten“ Wahl der ersten Clustermittelpunkte abhängig zu sein.

– Ergebnisvergleich k-Means mit Cosinus
Bildet man die Ergebnisse, die sich durch die Auswertung des k-Means-Clusterers mit Cosinus als Distanzmaß für das F_1-Maß für alle Datenteilmengen ergeben, in Diagrammen ab, so erhält man die Abbildung 5.16 auf S. 505.

Im Fall der TSF-Feature-Erzeugung „AllDoks“ ergibt sich über die drei Datenmengen für den k-Means-Clusterer mit Cosinus als Distanzmaß ein besseres Ergebnis für die TSF-Feature-Vektoren als für die einzelwortbasierten Feature-Vektoren. Im Durchschnitt unterscheidet sich das F_1-Maß für die beiden Verfahren um rund 4,47 Prozentpunkte, wobei der geringste Abstand 4 Prozentpunkte und der maximale Abstand 5,04 Prozentpunkte beträgt.

Bei der TSF-Feature-Erzeugung „SingleDoks“ liegen die Ergebnisse beider Verfahren sehr dicht beeinander. In 2 von 3 Fällen erreichen die TSF-Feature-Vektoren bessere Ergebnisse als die einzelwortbasierten Feature-Vektoren. Die erreichten Ergebnisse im Durchschnitt über die drei Datenmengen unterscheiden sich um rund 1,32 Prozentpunkte voneinander. Dabei beträgt der größte Abstand 2,24 Prozentpunkte und der kleinste Abstand 0,32 Prozentpunkte.

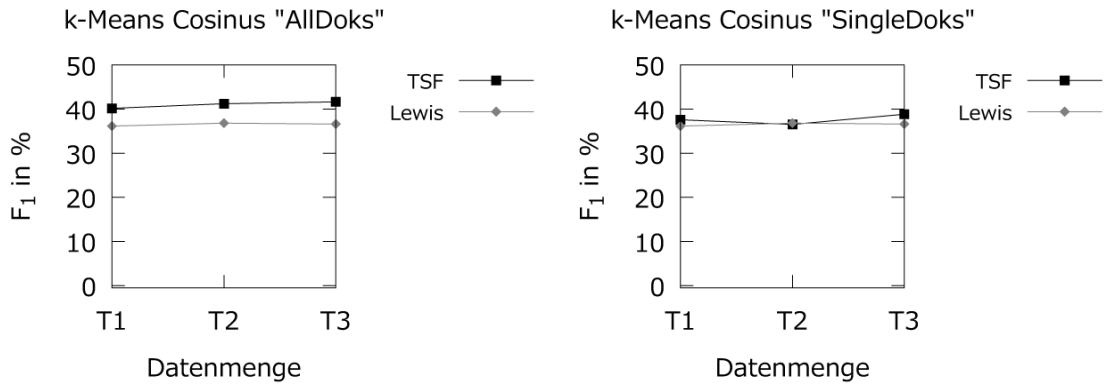


Abbildung 5.16: Ergebnis der Evaluation des durch den k-Means-Clusterer mit Cosinus erzeugten Clusterings für das F_1-Maß für das Experiment Clus3RV

Zusammengefasst ergibt sich für den k-Means-Clusterer mit Cosinus als Distanzmaß für das F_1-Maß folgendes Ergebnis:

* Sowohl bei „AllDoks“ als auch bei „SingleDoks“ erreichen die TSF-Feature-Vektoren bessere Ergebnisse als die einzelwortbasierten Feature-Vektoren.
* Nach diesem zweiten Teilexperiment von Experiment 9 scheint die TSF-Featurerzeugung, bei der alle Dokumente zur Feature-Erzeugung herangezogen werden besser zu sein als die Art, bei der alle Dokumente einzeln betrachtet werden.
* Auch in diesem Teilexperiment wird deutlich, wie sehr die Ergebnisse von der ersten Wahl der Clustermittelpunkte beeinflusst zu sein scheinen.

– Ergebnisvergleich GAAC

Bildet man die Ergebnisse, die sich durch die Auswertung des GAAC-Clusterers für das F_1-Maß für alle Datenteilmengen ergeben, in Diagrammen ab, so erhält man die Abbildung 5.17 auf dieser Seite.

Im Fall der TSF-Feature-Erzeugung „AllDoks“ ergibt sich in allen Fällen für den GAAC-Clusterer ein etwas besseres Ergebnis für die einzelwortbasierten Feature-Vektoren als für die TSF-Feature-Vektoren, wobei die Ergebnisse sehr nah zusammenliegen. Im Durchschnitt unterscheiden sich die Ergebnisse beider Verfahren um rund 1,82 Prozentpunkte, wobei der geringste Abstand 0,87 Prozentpunkte und der maximale Abstand 3,2 Prozentpunkte beträgt.

Betrachtet man die TSF-Feature-Erzeugung „SingleDoks“, so erreichen die einzelwortbasierten Feature-Vektoren in allen Fällen ebenfalls ein besseres Ergebnis, wobei die Ergebnisse sich kaum unterscheiden. So liegen sie für die einzelwortbasierten Feature-Vektoren im Durchschnitt 1,11 Prozentpunkte über den Ergebnissen für die TSF-Feature-Vektoren. Der kleinste Abstand beträgt dabei nur 0,04 Prozentpunkte und der größte Abstand 1,94 Prozentpunkte.

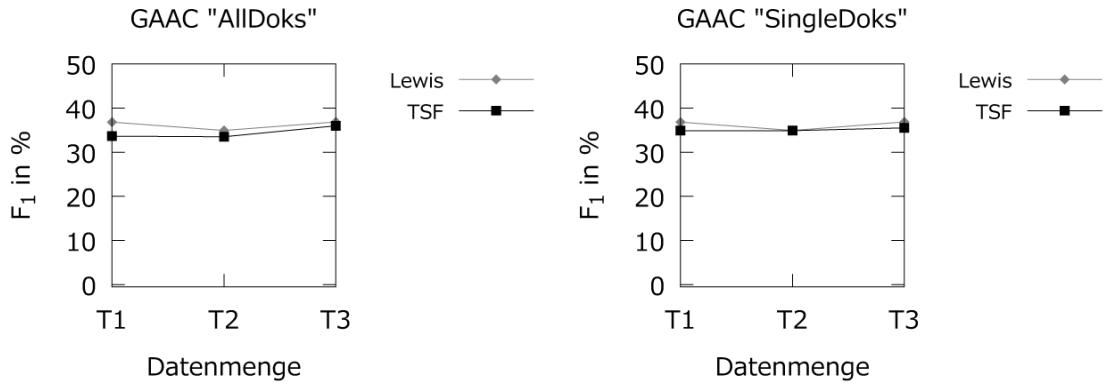


Abbildung 5.17: Ergebnis der Evaluation des durch den GAAC-Clusterer erzeugten Clusterings für das F_1-Maß für das Experiment Clus3RV

Zusammengefasst ergibt sich für den GAAC-Clusterer für das F_1-Maß folgendes Ergebnis:

* Bei beiden Arten der TSF-Feature-Erzeugung erreichen die einzelwortbasierten Feature-Vektoren etwas bessere Ergebnisse als die TSF-Feature-Vektoren.
* Nach diesem dritten Teilexperiment des Experiments 9 scheint die TSF-Featurerzeugung, bei der die Dokumente einzeln zur Feature-Erzeugung herangezogen werden, besser zu sein als die Art, bei der alle Dokumente betrachtet werden.

– Ergebnisvergleich über alle Algorithmen und Distanzmaße
Vergleicht man die Ergebnisse des einzelwortbasierten Verfahrens und des Verfahrens mit TSF-Features über alle Algorithmen miteinander, so stellt man Folgendes fest: Die TSF-Feature-Vektoren erreichen für das F_1-Maß in 4 von 6 Fällen ein besseres Ergebnis als die einzelwortbasierten Feature-Vektoren.

Dieses dritte Cluster-Experiment stützt also die Vermutung, dass wortübergreifende Features qualitativ bessere Ergebnisse beim Clustern von natürlichsprachlichen Dokumenten liefern als Verfahren, die natürlichsprachliche Dokumente in Einzelworte aufteilen und diese als Features für das Clustern benutzen.

Als Tabelle zusammengefasst ergibt sich für das dritte Cluster-Experiment folgendes Bild[1]:

Tabelle 5.24: Ergebnisse des Experiments Clus3RV

Verfahren	Algorithmus		
	k-Means Euklid	k-Means Cosinus	GAAC
„AllDoks“-Feature-Vektoren	✓	✓	
„SingleDoks“-Feature-Vektoren	✓	✓	
Lewis-Feature-Vektoren			✓✓

Damit erreichen die TSF-Feature-Vektoren, wenn man beide Erzeugungsarten zusammenfasst, in 4 von 6 Fällen ein besseres Ergebnis als die einzelwortbasierten Feature-Vektoren. Betrachtet man die beiden Erzeugungsarten der TSF-Features einzeln, so ergibt sich sowohl bei der Erzeugungsart „AllDoks“ als auch bei der Erzeugungsart „SingleDoks“ in 2 von 3 Fällen ein besseres Ergebnis der TSF-Feature-Vektoren. Insgesamt wird dieses dritte Cluster-Experiment also als Gleichstand zwischen den Ergebnissen aufgrund der unterschiedlichen TSF-Feature-Erzeugung gewertet.

5.3.3.4 Experiment 10

- ID
  Clus4RV

- Bezeichnung
  vektorbasiertes Region-Clustern von Daten der Reuters-Daten RCV1-v2 mit $F_{0,5}$ als Evaluationsmaß

1 Ein Häkchen zeigt das Verfahren an, das für den betreffenden Algorithmus die höchste Qualität über alle Datenmengen erreicht hat. Zwei Häkchen in der gleichen Spalte für Lewis bedeuten, dass die einzelwortbasierten Feature-Vektoren gegenüber beiden Arten der TSF-Feature-Erzeugung besser abgeschnitten haben.

- Daten
  Es werden die gleichen Daten verwendet wie in Experiment Clus1RV.
- Ähnlichkeit
  vektorbasiert
- Algorithmen
  Es werden die gleichen Algorithmen verwendet wie in Experiment Clus1RV.
- Anzahl Cluster
  4
- Evaluation
  $F_{0,5}$-Maß
- Ergebnisse[1]
  Die Ergebnisse für den k-Means-Clusterer mit euklidischem Distanzmaß befinden sich in den Tabellen 5.25 bis 5.27 auf Seite 508.
  Die Ergebnisse für den k-Means-Clusterer mit Cosinus-Distanzmaß befinden sich in den Tabellen 5.28 bis 5.30 auf den Seiten 509 bis 510.
  Die Ergebnisse für den GAAC-Clusterer befinden sich in Tabelle 5.31 auf S. 511.

Tabelle 5.25: Ergebnisse des Experiments Clus4RV für Datenmenge T1 und dem k-Means-Clusterer mit Euklid-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	$F_{0,5}$ in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	41,39
	Region	1	„SingleDoks“	33,26
Lewis-Feature-Vektoren	Region	1	-	36,77
TSF-Feature-Vektoren	Region	2	„AllDoks“	30,81
	Region	2	„SingleDoks“	31,38
Lewis-Feature-Vektoren	Region	2	-	36,78
TSF-Feature-Vektoren	Region	3	„AllDoks“	30,31
	Region	3	„SingleDoks“	31,33
Lewis-Feature-Vektoren	Region	3	-	33,19

1 In allen Tabellen werden die Ergebnisse auf zwei Stellen nach dem Komma gerundet. Das jeweils dunkel hinterlegte Tabellenfeld kennzeichnet beim k-Means-Clusterer das beste Ergebnis über die drei Durchläufe pro Feature-Erzeugungsmethode.

Tabelle 5.26: Ergebnisse des Experiments Clus4RV für Datenmenge T2 und dem k-Means-Clusterer mit Euklid-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	$F_{0,5}$ in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	32,98
	Region	1	„SingleDoks“	30,71
Lewis-Feature-Vektoren	Region	1	-	34,33
TSF-Feature-Vektoren	Region	2	„AllDoks“	31,70
	Region	2	„SingleDoks“	32,03
Lewis-Feature-Vektoren	Region	2	-	33,23
TSF-Feature-Vektoren	Region	3	„AllDoks“	35,00
	Region	3	„SingleDoks“	32,34
Lewis-Feature-Vektoren	Region	3	-	33,35

Tabelle 5.27: Ergebnisse des Experiments Clus4RV für Datenmenge T3 und dem k-Means-Clusterer mit Euklid-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	$F_{0,5}$ in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	30,93
	Region	1	„SingleDoks“	30,65
Lewis-Feature-Vektoren	Region	1	-	31,67
TSF-Feature-Vektoren	Region	2	„AllDoks“	34,31
	Region	2	„SingleDoks“	41,30
Lewis-Feature-Vektoren	Region	2	-	32,63
TSF-Feature-Vektoren	Region	3	„AllDoks“	34,18
	Region	3	„SingleDoks“	29,91
Lewis-Feature-Vektoren	Region	3	-	31,91

Tabelle 5.28: Ergebnisse des Experiments Clus4RV für Datenmenge T1 und dem k-Means-Clusterer mit Cosinus-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	$F_{0,5}$ in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	35,59
	Region	1	„SingleDoks“	30,46
Lewis-Feature-Vektoren	Region	1	-	33,43

wird auf der nächsten Seite fortgesetzt

Tabelle 5.28: Ergebnisse des Experiments Clus4RV für Datenmenge T1 und dem k-Means-Clusterer mit Cosinus-Distanzmaß (Fortsetzung)

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	$F_{0,5}$ in %
TSF-Feature-Vektoren	Region	2	„AllDoks“	29,74
	Region	2	„SingleDoks“	34,29
Lewis-Feature-Vektoren	Region	2	-	32,34
TSF-Feature-Vektoren	Region	3	„AllDoks“	34,43
	Region	3	„SingleDoks“	33,15
Lewis-Feature-Vektoren	Region	3	-	34,86

Tabelle 5.29: Ergebnisse des Experiments Clus4RV für Datenmenge T2 und dem k-Means-Clusterer mit Cosinus-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	$F_{0,5}$ in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	32,66
	Region	1	„SingleDoks“	32,35
Lewis-Feature-Vektoren	Region	1	-	33,28
TSF-Feature-Vektoren	Region	2	„AllDoks“	33,04
	Region	2	„SingleDoks“	32,27
Lewis-Feature-Vektoren	Region	2	-	33,46
TSF-Feature-Vektoren	Region	3	„AllDoks“	31,51
	Region	3	„SingleDoks“	32,90
Lewis-Feature-Vektoren	Region	3	-	33,81

Tabelle 5.30: Ergebnisse des Experiments Clus4RV für Datenmenge T3 und dem k-Means-Clusterer mit Cosinus-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	$F_{0,5}$ in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	30,40
	Region	1	„SingleDoks“	29,06
Lewis-Feature-Vektoren	Region	1	-	32,49
TSF-Feature-Vektoren	Region	2	„AllDoks“	34,22
	Region	2	„SingleDoks“	30,99
Lewis-Feature-Vektoren	Region	2	-	30,96

wird auf der nächsten Seite fortgesetzt

Tabelle 5.30: Ergebnisse des Experiments Clus4RV für Datenmenge T3 und dem k-Means-Clusterer mit Cosinus-Distanzmaß (Fortsetzung)

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	$F_{0,5}$ in %
TSF-Feature-Vektoren	Region	3	„AllDoks“	31,31
	Region	3	„SingleDoks“	33,68
Lewis-Feature-Vektoren	Region	3	-	32,41

Tabelle 5.31: Ergebnisse des Experiments Clus4RV für Datenmenge T1 bis T3 und dem GAAC-Clusterer

Verfahren	Daten-menge	Klassen-familie	Methode TSF-Features	$F_{0,5}$ in %
TSF-Feature-Vektoren	1	Region	„AllDoks“	31,19
	1	Region	„SingleDoks“	29,15
Lewis-Feature-Vektoren	1	Region	-	34,60
TSF-Feature-Vektoren	2	Region	„AllDoks“	29,45
	2	Region	„SingleDoks“	29,64
Lewis-Feature-Vektoren	2	Region	-	33,15
TSF-Feature-Vektoren	3	Region	„AllDoks“	32,26
	3	Region	„SingleDoks“	29,89
Lewis-Feature-Vektoren	3	Region	-	34,38

- Interpretation der Ergebnisse

  - Einführung
    Im Folgenden werden die vorgestellten Ergebnisse für die auf den Suffix Arrays basierenden Vektoren und für die Lewis-Feature-Vektoren miteinander verglichen. Dabei werden zunächst die besten Ergebnisse pro Algorithmus und Distanzmaß über die drei Datenmengen miteinander verglichen und abschließend erfolgt ein Vergleich für das gesamte Experiment.

  - Ergebnisvergleich k-Means mit Euklid
    Bildet man die Ergebnisse, die sich durch die Auswertung des k-Means-Clusterers mit Euklid als Distanzmaß für das $F_{0,5}$-Maß für alle Datenmengen ergeben, in Diagrammen ab, so erhält man Folgendes:

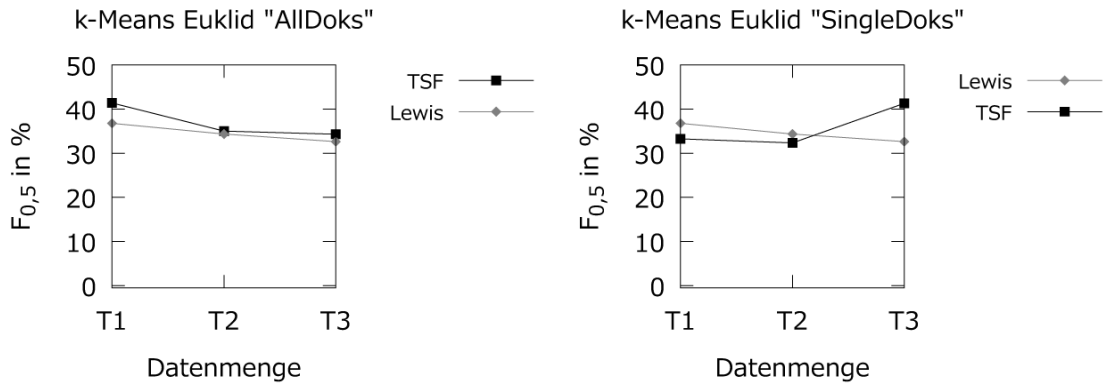


Abbildung 5.18: Ergebnis der Evaluation des durch den k-Means-Clusterer mit Euklid erzeugten Clusterings für das $F_{0,5}$-Maß für das Experiment Clus4RV

Im Fall der TSF-Feature-Erzeugung „AllDoks“ ergibt sich über die drei Datenmengen für den k-Means-Clusterer mit euklidischem Distanzmaß, dass die TSF-Feature-Vektoren bessere Ergebnisse erzielen als die einzelwortbasierten Feature-Vektoren. Im Durchschnitt unterscheiden sich die Ergebnisse um rund 2,32 Prozentpunkte, wobei der minimale Abstand 0,67 Prozentpunkte und der maximale Abstand 4,61 Prozentpunkte beträgt.

Betrachtet man dagegen die TSF-Feature-Erzeugung „SingleDoks“, so erreichen die TSF-Feature-Vektoren in 1 von 3 Fällen ein besseres Ergebnis für das $F_{0,5}$-Maß. So unterscheiden sich die erreichten Ergebnisse im Durchschnitt über die drei Datenmengen um rund 4,73 Prozentpunkte voneinander. Dabei wird der größte Abstand von 8,68 Prozentpunkten bei Datenmenge T3 erreicht, also in dem Fall, dass die TSF-Feature-Vektoren ein besseres Ergebnis erreichen, und der kleinste von 1,99 bei Datenmenge T2 mit einem besseren Ergebnis für die einzelwortbasierten Feature-Vektoren.

Zusammengefasst ergibt sich für den k-Means-Clusterer mit euklidischem Distanzmaß für das $F_{0,5}$-Maß folgendes Ergebnis:

* Bei „AllDoks“ erreichen die TSF-Feature-Vektoren in allen Fällen ein besseres Ergebnis als die einzelwortbasierten Feature-Vektoren.
* Bei „SingleDoks“ erreichen die einzelwortbasierten Feature-Vektoren für zwei Datenmengen bessere Ergebnisse als die TSF-Feature-Vektoren.

  - Insgesamt ergibt sich damit, dass in diesem Teilexperiment des Experiments 10 sich ein Gleichstand zwischen den beiden Verfahren ergibt.
  - Nach diesem ersten Teilexperiment von Experiment 10 scheint die TSF-Featurerzeugung, bei der alle Dokumente zur Feature-Erzeugung herangezogen werden, besser zu sein als die Art, bei der jedes Dokument einzeln betrachtet wird.
  - Auch bei diesem Teilexperiment scheinen die Ergebnisse von einer „guten“ Wahl der ersten Clustermittelpunkte abhängig zu sein.

- Ergebnisvergleich k-Means mit Cosinus
  Bildet man die Ergebnisse, die sich durch die Auswertung des k-Means-Clusterers mit Cosinus als Distanzmaß für das $F_{0,5}$-Maß für alle Datenteilmengen ergeben, in Diagrammen ab, so erhält man die Abbildung 5.19 auf S. 514.

  Im Fall der TSF-Feature-Erzeugung „AllDoks“ ergibt sich über die drei Datenmengen für den k-Means-Clusterer mit Cosinus als Distanzmaß kaum ein Unterschied zwischen den Ergebnissen beider Verfahren. Im Durchschnitt unterscheidet sich das $F_{0,5}$-Maß für die beiden Verfahren um rund 1,08 Prozentpunkte, wobei der geringste Abstand 0,73 Prozentpunkte und der maximale Abstand 1,74 Prozentpunkte beträgt. Insgesamt werden in 2 von 3 Fällen etwas bessere Ergebnisse für die TSF-Feature-Vektoren erzielt.

  Bei der TSF-Feature-Erzeugung „SingleDoks“ ergibt sich ein umgekehrtes Bild. In 2 von 3 Fällen erreichen hier die einzelwortbasierten Feature-Vektoren bessere Ergebnisse als die TSF-Feature-Vektoren. Die erreichten Ergebnisse unterscheiden sich im Durchschnitt über die drei Datenmengen um rund 0,89 Prozentpunkte voneinander und liegen damit noch enger zusammen als bei den „AllDoks“. Der größte Abstand bei den „SingleDoks“ liegt bei 1,2 Prozentpunkten und der kleinste Abstand bei 0,57 Prozentpunkten. Zusammengefasst ergibt sich für den k-Means-Clusterer mit Cosinus als Distanzmaß für das $F_{0,5}$-Maß folgendes Ergebnis:

  - Bei „AllDoks“ erreichen die TSF-Feature-Vektoren bessere Ergebnisse als die einzelwortbasierten Feature-Vektoren.
  - Bei „SingleDoks“ ist es genau umgekehrt.
  - Insgesamt ergibt sich also ein Gleichstand zwischen den beiden Verfahren.
  - Nach diesem zweiten Teilexperiment des Experiments 10 scheint die TSF-Featurerzeugung, bei der alle Dokumente zur Feature-Erzeugung

herangezogen werden, besser zu sein als die Art, bei der alle Dokumente einzeln betrachtet werden.

* Auch in diesem Teilexperiment wird deutlich, wie sehr die Ergebnisse von der ersten Wahl der Clusterzentren beeinflusst zu sein scheinen.

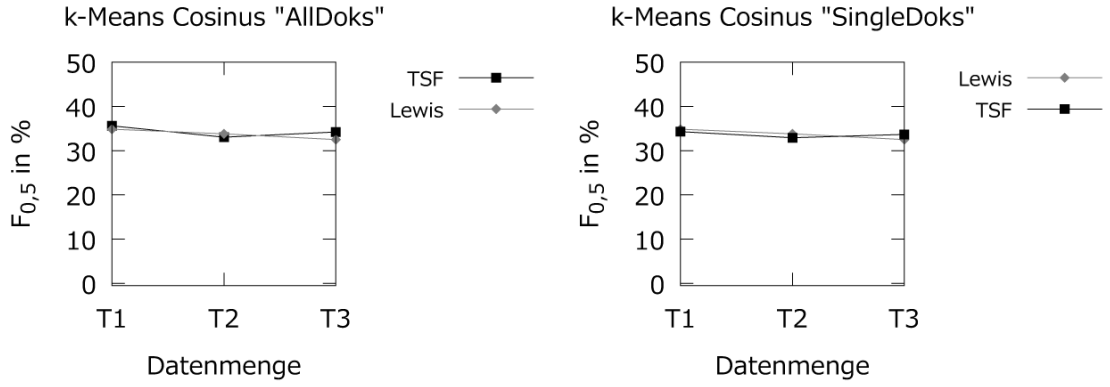


Abbildung 5.19: Ergebnis der Evaluation des durch den k-Means-Clusterer mit Cosinus erzeugten Clusterings für das $F_{0,5}$-Maß für das Experiment Clus4RV

– Ergebnisvergleich GAAC
Bildet man die Ergebnisse, die sich durch die Auswertung des GAAC-Clusterers für das $F_{0,5}$-Maß für alle Datenteilmengen ergeben, in Diagrammen ab, so erhält man die Abbildung 5.20 auf S. 515.

Im Fall der TSF-Feature-Erzeugung „AllDoks“ ergibt sich in allen Fällen für den GAAC-Clusterer ein etwas besseres Ergebnis für die einzelwortbasierten Feature-Vektoren als für die TSF-Feature-Vektoren. Im Durchschnitt unterscheiden sich die Ergebnisse beider Verfahren um rund 3,08 Prozentpunkte, wobei der geringste Abstand 2,13 Prozentpunkte und der maximale Abstand 3,7 Prozentpunkte beträgt.

Betrachtet man die TSF-Feature-Erzeugung „SingleDoks“, so erreichen die einzelwortbasierten Feature-Vektoren in allen Fällen ebenfalls ein besseres Ergebnis. Die Ergebnisse liegen für die einzelwortbasierten Feature-Vektoren im Durchschnitt 4,49 Prozentpunkte über den Ergebnissen für die TSF-Feature-Vektoren. Der kleinste Abstand beträgt dabei 3,51 Prozentpunkte und der größte Abstand 5,45 Prozentpunkte.

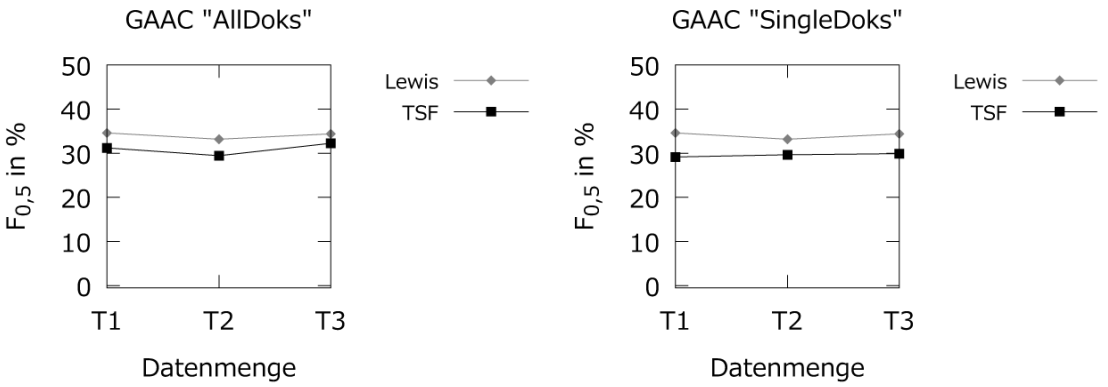


Abbildung 5.20: Ergebnis der Evaluation des durch den GAAC-Clusterer erzeugten Clusterings für das $F_{0,5}$-Maß für das Experiment Clus4RV

Zusammengefasst ergibt sich für den GAAC-Clusterer für das $F_{0,5}$-Maß folgendes Ergebnis:

* Bei beiden Arten der TSF-Feature-Erzeugung erreichen die einzelwortbasierten Feature-Vektoren etwas bessere Ergebnisse als die TSF-Feature-Vektoren.
* Nach diesem dritten Teilexperiment des Experiments 10 scheint die TSF-Featurerzeugung, bei der alle Dokumente zur Feature-Erzeugung herangezogen werden, besser zu sein als die Art, bei der jedes Dokument einzeln betrachtet wird.

– Ergebnisvergleich über alle Algorithmen und Distanzmaße
Vergleicht man die Ergebnisse des einzelwortbasierten Verfahrens und des Verfahrens mit TSF-Features über alle Algorithmen miteinander, so stellt man Folgendes fest: Die einzelwortbasierten Feature-Vektoren erreichen für das $F_{0,5}$-Maß in 4 von 6 Fällen ein besseres Ergebnis als die TSF-Feature-Vektoren.

Dieses vierte Cluster-Experiment widerlegt also die Vermutung, dass wortübergreifende Features qualitativ bessere Ergebnisse beim Clustern von natürlichsprachlichen Dokumenten liefern als Verfahren, die natürlichsprachliche Dokumente in Einzelworte aufteilen und diese als Features für das Clustern benutzen.

Als Tabelle zusammengefasst ergibt sich für das Experiment 10 folgendes Bild[1]:

Tabelle 5.32: Ergebnisse des Experiments Clus4RV

Verfahren	Algorithmus		
	k-Means Euklid	k-Means Cosinus	GAAC
„AllDoks“-Feature-Vektoren	✓	✓	
„SingleDoks“-Feature-Vektoren			
Lewis-Feature-Vektoren	✓	✓	✓✓

Damit erreichen die TSF-Feature-Vektoren, wenn man beide Erzeugungsarten zusammenfasst, in 2 von 6 Fällen ein besseres Ergebnis als die einzelwortbasierten Feature-Vektoren. Betrachtet man die beiden Erzeugungsarten der TSF-Features einzeln, so ergibt sich bei der Erzeugungsart „AllDoks“ in 2 von 3 Fällen ein besseres Ergebnis der TSF-Feature-Vektoren. Die Erzeugungsart „SingleDoks“ erreicht in keinem Fall ein besseres Ergebnis als die einzelwortbasierten Feature-Vektoren. Insgesamt weist dieses vierte Experiment für das Clustern darauf hin, dass die Feature-Erzeugung bei der alle Dokumente gemeinsam betrachtet werden, bessere Ergebnisse erzielt als die Feature-Erzeugung, bei der jedes Dokument einzeln betrachtet wird.

5.3.3.5 Experiment 11

- ID
  Clus5RV

- Bezeichnung
  vektorbasiertes Region-Clustern von Daten der Reuters-Daten RCV1-v2 mit F_2 als Evaluationsmaß

- Daten
  Es werden die gleichen Daten verwendet wie in Experiment Clus1RV.

1 Ein Häkchen zeigt das Verfahren an, das für den betreffenden Algorithmus die höchste Qualität über alle Datenmengen erreicht hat. Zwei Häkchen in der gleichen Spalte für Lewis bedeuten, dass die einzelwortbasierten Feature-Vektoren gegenüber beiden Arten der TSF-Feature-Erzeugung besser abgeschnitten haben.

- Ähnlichkeit
  vektorbasiert

- Algorithmen
  Es werden die gleichen Algorithmen verwendet wie in Experiment Clus1RV.

- Anzahl Cluster
  4

- Evaluation
  F_2-Maß

- Ergebnisse[1]
  Die Ergebnisse für den k-Means-Clusterer mit euklidischem Distanzmaß befinden sich in den Tabellen 5.33 bis 5.35 auf dieser Seite bis Seite518.
  Die Ergebnisse für den k-Means-Clusterer mit Cosinus-Distanzmaß befinden sich in den Tabellen 5.36 bis 5.38 auf den Seiten 518 bis 519.
  Die Ergebnisse für den GAAC-Clusterer befinden sich in Tabelle 5.39 auf S. 520.

Tabelle 5.33: Ergebnisse des Experiments Clus5RV für Datenmenge T1 und dem k-Means-Clusterer mit Euklid-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	F_2 in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	47,60
	Region	1	„SingleDoks“	37,53
Lewis-Feature-Vektoren	Region	1	-	43,57
TSF-Feature-Vektoren	Region	2	„AllDoks“	47,98
	Region	2	„SingleDoks“	42,12
Lewis-Feature-Vektoren	Region	2	-	40,88
TSF-Feature-Vektoren	Region	3	„AllDoks“	44,95
	Region	3	„SingleDoks“	41,16
Lewis-Feature-Vektoren	Region	3	-	39,32

1 In allen Tabellen werden die Ergebnisse auf zwei Stellen nach dem Komma gerundet. Das jeweils dunkel hinterlegte Tabellenfeld kennzeichnet beim k-Means-Clusterer das beste Ergebnis über die drei Durchläufe pro Feature-Erzeugungsmethode.

Tabelle 5.34: Ergebnisse des Experiments Clus5RV für Datenmenge T2 und dem k-Means-Clusterer mit Euklid-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	F_2 in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	39,62
	Region	1	„SingleDoks“	57,61
Lewis-Feature-Vektoren	Region	1	-	38,91
TSF-Feature-Vektoren	Region	2	„AllDoks“	44,48
	Region	2	„SingleDoks“	43,85
Lewis-Feature-Vektoren	Region	2	-	39,82
TSF-Feature-Vektoren	Region	3	„AllDoks“	52,75
	Region	3	„SingleDoks“	36,26
Lewis-Feature-Vektoren	Region	3	-	39,55

Tabelle 5.35: Ergebnisse des Experiments Clus5RV für Datenmenge T3 und dem k-Means-Clusterer mit Euklid-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	F_2 in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	45,17
	Region	1	„SingleDoks“	38,23
Lewis-Feature-Vektoren	Region	1	-	49,12
TSF-Feature-Vektoren	Region	2	„AllDoks“	41,12
	Region	2	„SingleDoks“	50,35
Lewis-Feature-Vektoren	Region	2	-	46,30
TSF-Feature-Vektoren	Region	3	„AllDoks“	51,67
	Region	3	„SingleDoks“	43,94
Lewis-Feature-Vektoren	Region	3	-	41,45

Tabelle 5.36: Ergebnisse des Experiments Clus5RV für Datenmenge T1 und dem k-Means-Clusterer mit Cosinus-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	F_2 in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	46,03
	Region	1	„SingleDoks“	40,28
Lewis-Feature-Vektoren	Region	1	-	36,56

wird auf der nächsten Seite fortgesetzt

Tabelle 5.36: Ergebnisse des Experiments Clus5RV für Datenmenge T1 und dem k-Means-Clusterer mit Cosinus-Distanzmaß (Fortsetzung)

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	F_2 in %
TSF-Feature-Vektoren	Region	2	„AllDoks“	46,36
	Region	2	„SingleDoks“	41,49
Lewis-Feature-Vektoren	Region	2	-	40,96
TSF-Feature-Vektoren	Region	3	„AllDoks“	47,50
	Region	3	„SingleDoks“	39,15
Lewis-Feature-Vektoren	Region	3	-	36,86

Tabelle 5.37: Ergebnisse des Experiments Clus5RV für Datenmenge T2 und dem k-Means-Clusterer mit Cosinus-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	F_2 in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	53,57
	Region	1	„SingleDoks“	39,48
Lewis-Feature-Vektoren	Region	1	-	39,43
TSF-Feature-Vektoren	Region	2	„AllDoks“	54,73
	Region	2	„SingleDoks“	41,03
Lewis-Feature-Vektoren	Region	2	-	37,36
TSF-Feature-Vektoren	Region	3	„AllDoks“	52,30
	Region	3	„SingleDoks“	41,02
Lewis-Feature-Vektoren	Region	3	-	40,46

Tabelle 5.38: Ergebnisse des Experiments Clus5RV für Datenmenge T3 und dem k-Means-Clusterer mit Cosinus-Distanzmaß

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	F_2 in %
TSF-Feature-Vektoren	Region	1	„AllDoks“	51,77
	Region	1	„SingleDoks“	38,15
Lewis-Feature-Vektoren	Region	1	-	41,86
TSF-Feature-Vektoren	Region	2	„AllDoks“	53,11
	Region	2	„SingleDoks“	38,72
Lewis-Feature-Vektoren	Region	2	-	35,80

wird auf der nächsten Seite fortgesetzt

Tabelle 5.38: Ergebnisse des Experiments Clus5RV für Datenmenge T3 und dem k-Means-Clusterer mit Cosinus-Distanzmaß (Fortsetzung)

Verfahren	Klassen-familie	Durchlauf	Methode TSF-Features	F_2 in %
TSF-Feature-Vektoren	Region	3	„AllDoks“	50,86
	Region	3	„SingleDoks“	45,82
Lewis-Feature-Vektoren	Region	3	-	41,31

Tabelle 5.39: Ergebnisse des Experiments Clus5RV für Datenmenge T1 bis T3 und dem GAAC-Clusterer

Verfahren	Daten-menge	Klassen-familie	Methode TSF-Features	F_2 in %
TSF-Feature-Vektoren	1	Region	„AllDoks“	36,43
	1	Region	„SingleDoks“	43,38
Lewis-Feature-Vektoren	1	Region	-	39,32
TSF-Feature-Vektoren	2	Region	„AllDoks“	38,87
	2	Region	„SingleDoks“	42,36
Lewis-Feature-Vektoren	2	Region	-	36,87
TSF-Feature-Vektoren	3	Region	„AllDoks“	40,66
	3	Region	„SingleDoks“	43,71
Lewis-Feature-Vektoren	3	Region	-	39,68

- Interpretation der Ergebnisse

  – Einführung
  Im Folgenden werden die vorgestellten Ergebnisse für die auf den Suffix Arrays basierenden Vektoren und für die Lewis-Feature-Vektoren miteinander verglichen. Dabei werden zunächst die besten Ergebnisse pro Algorithmus und Distanzmaß über die drei Datenmengen miteinander verglichen und abschließend erfolgt ein Vergleich für das gesamte Experiment.

  – Ergebnisvergleich k-Means mit Euklid
  Bildet man die Ergebnisse, die sich durch die Auswertung des k-Means-Clusterers mit Euklid als Distanzmaß für das F_2-Maß für alle Datenmengen ergeben, in Diagrammen ab, so erhält man Folgendes:

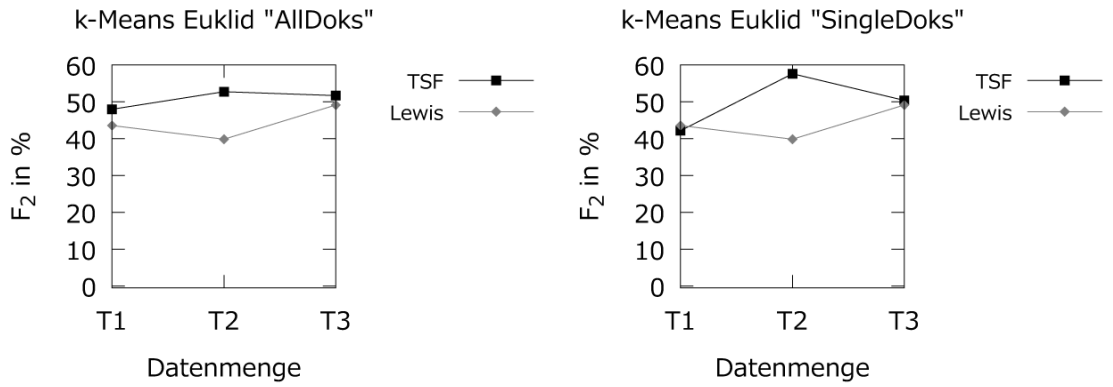


Abbildung 5.21: Ergebnis der Evaluation des durch den k-Means-Clusterer mit Euklid erzeugten Clusterings für das F_2-Maß für das Experiment Clus5RV

Im Fall der TSF-Feature-Erzeugung „AllDoks“ ergibt sich über die drei Datenmengen für den k-Means-Clusterer mit euklidischem Distanzmaß, dass die TSF-Feature-Vektoren bessere Ergebnisse erzielen als die einzelwortbasierten Feature-Vektoren. Im Durchschnitt unterscheiden sich die Ergebnisse um rund 6,63 Prozentpunkte, wobei der minimale Abstand 2,55 Prozentpunkte und der maximale Abstand 12,92 Prozentpunkte beträgt.

Betrachtet man dagegen die TSF-Feature-Erzeugung „SingleDoks“, so erreichen die TSF-Feature-Vektoren in 2 von 3 Fällen ein besseres Ergebnis für das F_2-Maß. So unterscheiden sich die erreichten Ergebnisse im Durchschnitt über die drei Datenmengen um rund 6,82 Prozentpunkte voneinander. Dabei wird der größte Abstand von 17,78 Prozentpunkten bei Datenmenge T2 erreicht, also in dem Fall, dass die TSF-Feature-Vektoren ein besseres Ergebnis erreichen, und der kleinste von 1,23 bei Datenmenge T3 ebenfalls mit einem besseren Ergebnis für die TSF-Feature-Vektoren.

Zusammengefasst ergibt sich für den k-Means-Clusterer mit euklidischem Distanzmaß für das F_2-Maß folgendes Ergebnis:

* Bei „AllDoks“ erreichen die TSF-Feature-Vektoren in allen Fällen ein besseres Ergebnis als die einzelwortbasierten Feature-Vektoren.
* Bei „SingleDoks“ erreichen die TSF-Feature-Vektoren für zwei Datenmengen bessere Ergebnisse als die einzelwortbasierten Feature-Vektoren.

  - Insgesamt ergibt sich damit, dass in diesem Teilexperiment die TSF-Feature-Vektoren bessere Ergebnisse erzielen, als die einzelwortbasierten Feature-Vektoren.
  - Nach diesem ersten Teilexperiment von Experiment 11 scheint die TSF-Featureerzeugung, bei der alle Dokumente zur Feature-Erzeugung herangezogen werden besser zu sein als die Art, bei der jedes Dokument einzeln betrachtet wird.
  - Auch bei diesem Teilexperiment scheinen die Ergebnisse von einer „guten“ Wahl der ersten Clustermittelpunkte abhängig zu sein.

– Ergebnisvergleich k-Means mit Cosinus
  Bildet man die Ergebnisse, die sich durch die Auswertung des k-Means-Clusterers mit Cosinus als Distanzmaß für das F_2-Maß für alle Datenteilmengen ergeben, in Diagrammen ab, so erhält man die Abbildung 5.22 auf S. 523.

  Im Fall der TSF-Feature-Erzeugung „AllDoks“ ergibt sich über die drei Datenmengen für den k-Means-Clusterer mit Cosinus als Distanzmaß ein deutlicher Unterschied zwischen den Ergebnissen beider Verfahren. Im Durchschnitt unterscheidet sich das F_2-Maß für die beiden Verfahren um rund 10,68 Prozentpunkte, wobei der geringste Abstand 6,54 Prozentpunkte und der maximale Abstand 14,27 Prozentpunkte beträgt. Insgesamt werden in allen Fällen bessere Ergebnisse für die TSF-Feature-Vektoren erzielt.

  Bei der TSF-Feature-Erzeugung „SingleDoks“ ergibt sich kaum ein Unterschied zwischen den Ergebnissen der beiden Verfahren. In allen Fällen erreichen hier die TSF-Feature-Vektoren geringfügig bessere Ergebnisse als die einzelwortbasierten Feature-Vektoren. Die erreichten Ergebnisse im Durchschnitt über die drei Datenmengen unterscheiden sich um rund 1,69 Prozentpunkte voneinander. Dabei beträgt der größte Abstand 3,96 Prozentpunkte und der kleinste Abstand 0,53 Prozentpunkte.

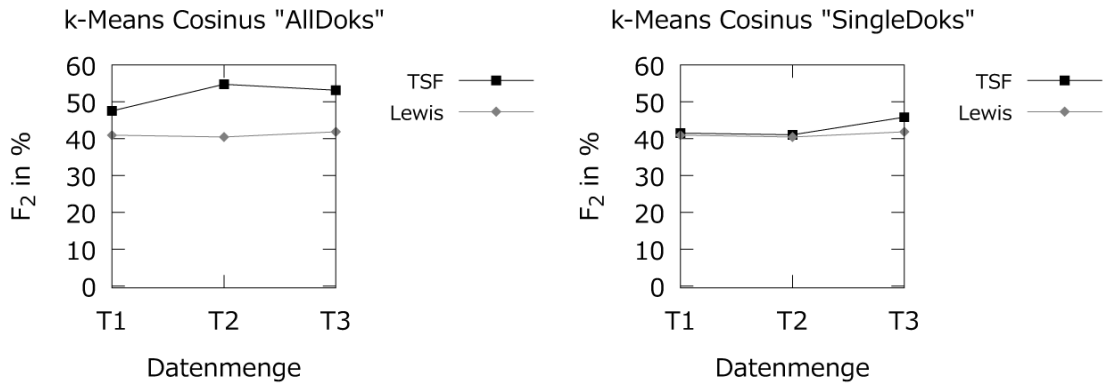


Abbildung 5.22: Ergebnis der Evaluation des durch den k-Means-Clusterer mit Cosinus erzeugten Clusterings für das F_2-Maß für das Experiment Clus5RV

Zusammengefasst ergibt sich für den k-Means-Clusterer mit Cosinus als Distanzmaß für das F_2-Maß folgendes Ergebnis:

* Bei „AllDoks“ erreichen die TSF-Feature-Vektoren bessere Ergebnisse als die einzelwortbasierten Feature-Vektoren.
* Bei „SingleDoks“ ist es genauso, aber die Ergebnisse liegen dichter beieinander.
* Insgesamt ergibt sich also, dass die TSF-Feature-Vektoren bessere Ergebnisse erzielen als die einzelwortbasierten Feature-Vektoren.
* Nach diesem zweiten Teilexperiment des Experiments 11 scheint die TSF-Featurerzeugung, bei der alle Dokumente zur Feature-Erzeugung herangezogen werden, besser zu sein als die Art, bei der alle Dokumente einzeln betrachtet werden.
* Auch in diesem Teilexperiment wird deutlich, wie sehr die Ergebnisse von der ersten Wahl der Clustermittelpunkte beeinflusst zu sein scheinen.

– Ergebnisvergleich GAAC
Bildet man die Ergebnisse, die sich durch die Auswertung des GAAC-Clusterers für das F_2-Maß für alle Datenteilmengen ergeben, in Diagrammen ab, so erhält man Folgendes:

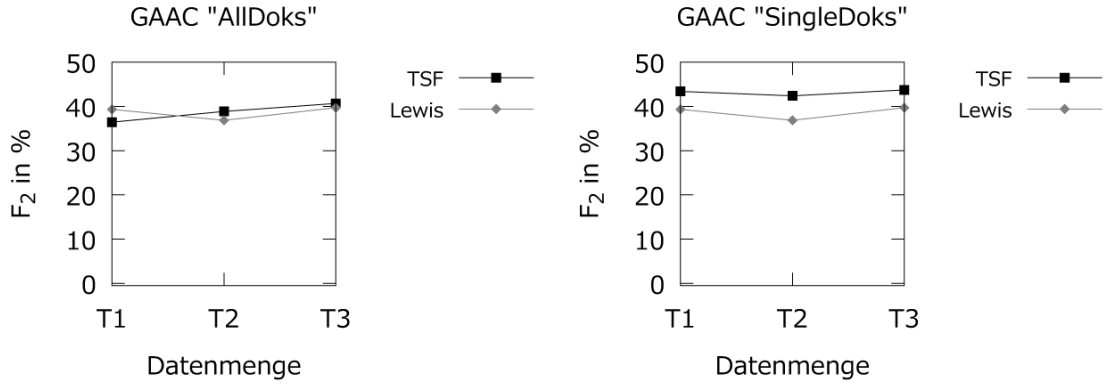


Abbildung 5.23: Ergebnis der Evaluation des durch den GAAC-Clusterer erzeugten Clusterings für das F_2-Maß für das Experiment Clus5RV

Im Fall der TSF-Feature-Erzeugung „AllDoks“ ergibt sich in 2 von 3 Fällen für den GAAC-Clusterer ein besseres Ergebnis für die TSF-Feature-Vektoren als für die einzelwortbasierten Feature-Vektoren. Im Durchschnitt unterscheiden sich die Ergebnisse beider Verfahren um rund 1,96 Prozentpunkte, wobei der geringste Abstand 0,99 Prozentpunkte und der maximale Abstand 2,89 Prozentpunkte beträgt.

Betrachtet man die TSF-Feature-Erzeugung „SingleDoks“, so erreichen die TSF-Feature-Vektoren in allen Fällen ein besseres Ergebnis. Die Ergebnisse liegen für die TSF-Feature-Vektoren im Durchschnitt 4,52 Prozentpunkte über den Ergebnissen für die einzelwortbasierten Feature-Vektoren. Der kleinste Abstand beträgt dabei 4,03 Prozentpunkte und der größte Abstand 4,48 Prozentpunkte.

Zusammengefasst ergibt sich für den GAAC-Clusterer für das F_2-Maß folgendes Ergebnis:

* Bei beiden Arten der TSF-Feature-Erzeugung erreichen die TSF-Feature-Vektoren bessere Ergebnisse als die einzelwortbasierten Feature-Vektoren.

  - Nach diesem dritten Teilexperiment des Experiments 11 scheint die TSF-Feature-Erzeugung, bei der alle Dokumente einzeln bei der Feature-Erzeugung betrachtet werden, besser zu sein als die Art, bei der alle Dokumente gemeinsam betrachtet werden.

- Ergebnisvergleich über alle Algorithmen und Distanzmaße
  Vergleicht man die Ergebnisse des einzelwortbasierten Verfahrens und des Verfahrens mit TSF-Features über alle Algorithmen miteinander, so stellt man Folgendes fest: Die TSF-Feature-Vektoren erreichen für das F_2-Maß in allen 6 Fällen ein besseres Ergebnis als die einzelwortbasierten Feature-Vektoren.

  Dieses fünfte Experiment für das Clustern stützt also die Vermutung, dass wortübergreifende Features qualitativ bessere Ergebnisse beim Clustern von natürlichsprachlichen Dokumenten liefern als Verfahren, die natürlichsprachliche Dokumente in Einzelworte aufteilen und diese als Features für das Clustern benutzen.

  Als Tabelle zusammengefasst ergibt sich für das fünfte Cluster-Experiment folgendes Bild[1]:

Tabelle 5.40: Ergebnisse des Experiments Clus5RV

Verfahren	Algorithmus		
	k-Means Euklid	k-Means Cosinus	GAAC
„AllDoks“-Feature-Vektoren	✓	✓	✓
„SingleDoks“-Feature-Vektoren	✓	✓	✓
Lewis-Feature-Vektoren			

  Damit erreichen die TSF-Feature-Vektoren, wenn man beide Erzeugungsarten zusammenfasst, in allen 6 Fällen ein besseres Ergebnis als die einzelwortbasierten Feature-Vektoren. Betrachtet man die beiden Erzeugungsarten der TSF-Features einzeln, so ergibt sich sowohl bei der Erzeugungsart „AllDoks“ in 3 von 3 Fällen ein besseres Ergebnis der TSF-Feature-Vektoren, als auch

1 Ein Häkchen zeigt das Verfahren an, das für den betreffenden Algorithmus die höchste Qualität über alle Datenmengen erreicht hat. Zwei Häkchen in der gleichen Spalte für Lewis bedeuten, dass die einzelwortbasierten Feature-Vektoren gegenüber beiden Arten der TSF-Feature-Erzeugung besser abgeschnitten haben.

bei der Erzeugungsart „SingleDoks“. Insgesamt wird dieses fünfte Experiment für das Clustern also als Gleichstand zwischen den Ergebnissen aufgrund der unterschiedlichen TSF-Feature-Erzeugung gewertet.

5.3.3.6 Vergleich der Ergebnisse des Clusterns über alle Experimente

Fasst man die Ergebnisse aller Experimente des Clusterns von natürlichsprachlichen Dokumenten noch einmal überblicksartig zusammen, so ergibt sich Folgendes:

- In rund 53% der durchgeführten Experimente erreichen die wortübergreifenden Feature-Vektoren bessere Clusterergebnisse als die einzelwortbasierten Feature-Vektoren. Das ist zwar etwas mehr als die Hälfte aller Fälle, jedoch lässt sich damit nicht eindeutig sagen, dass die TSF-Feature-Vektoren bessere Ergebnisse liefern als die einzelwortbasierten Feature-Vektoren. Vielmehr muss man in Bezug auf das Clustern von Dokumenten weiter differenzieren, in welchen Fällen die TSF-Feature-Vektoren eher geeignet sind und in welchen Fällen die einzelwortbasierten.

  Alle Evaluationsmaße, die in den Experimenten verwendet werden, werten die TP, FP, FN und TN aus, aber auf unterschiedliche Art und Weise. Da sich aus den Auswertungen ein annähernder Gleichstand zwischen den beiden Verfahren in den durchgeführten Experimenten herauskristallisiert hat, erfolgt im Folgenden eine Betrachtung der genannten Anzahlen, die den Maßen zu Grunde liegen.

  Stellt man die Anzahlen für den k-Means-Algorithmus mit euklidischem Distanzmaß grafisch dar, so erhält man die Diagramme[1] 5.24 auf S. 528.

  Daran abzulesen ist:

  1. Bei den beiden Anzahlen, die immer positiv[2] in die Evaluationsmaße einfließen - die TP und die TN - sind in beiden Fällen die Ergebnisse mit TSF-Feature-Vektoren besser als mit den einzelwortbasierten Feature-Vektoren. Bei den TP werden die höchsten Anzahlen über alle drei Datenmengen durch die „AllDoks“-Feature-Vektoren erreicht. Das Ergebnis bei den TN ist nicht so eindeutig, jedoch zeigen die „SingleDoks“-Feature-Vektoren sehr konstante Ergebnisse, während die Ergebnisse der beiden

1 Es werden die Werte betrachtet, die zum besten Durchlauf für die jeweilige Datenmenge geführt haben. Bei einem Gleichstand zwischen zwei Durchläufen wird ein Durchlauf zufällig ausgewählt.

2 Positiv heißt im Zusammenhang mit Evaluationsmaßen, dass die Anzahl oder die Anzahlen dazu führen, dass der Wert des Evaluationsmaßes steigt, also das Evaluationsergebnis „besser“ wird.

anderen Verfahren stark schwanken, so dass im Schnitt über alle drei Datenmengen die „SingleDoks“-Feature-Vektoren die höchsten Anzahlen für TN erreichen.

2. Jedoch sind auch bei den beiden Anzahlen, die negativ[1] in die Evaluationsmaße einfließen - die FP und die FN - in beiden Fällen die höchsten Anzahlen bei den TSF-Feature-Vektoren zu finden. So erreichen die „AllDoks“-Feature-Vektoren bei den FP in 2 von 3 Fällen die höchste Anzahl, daher wird das insgesamt als Verfahren mit den höchsten Anzahlen für die FP gezählt. Darauf folgen die einzelwortbasierten Feature-Vektoren und abschließend die „SingleDoks“-Feature-Vektoren. Bei den FN ist dagegen die Reihenfolge genau umgekehrt.
3. Die einzelwortbasierten Feature-Vektoren erreichen bei dieser Art der Betrachtung immer die Mitte der drei Verfahren.

1 Negativ heißt im Zusammenhang mit Evaluationsmaßen, dass die Anzahl oder die Anzahlen dazu führen, dass der Wert des Evaluationsmaßes sinkt, also das Evaluationsergebnis „schlechter“ wird.

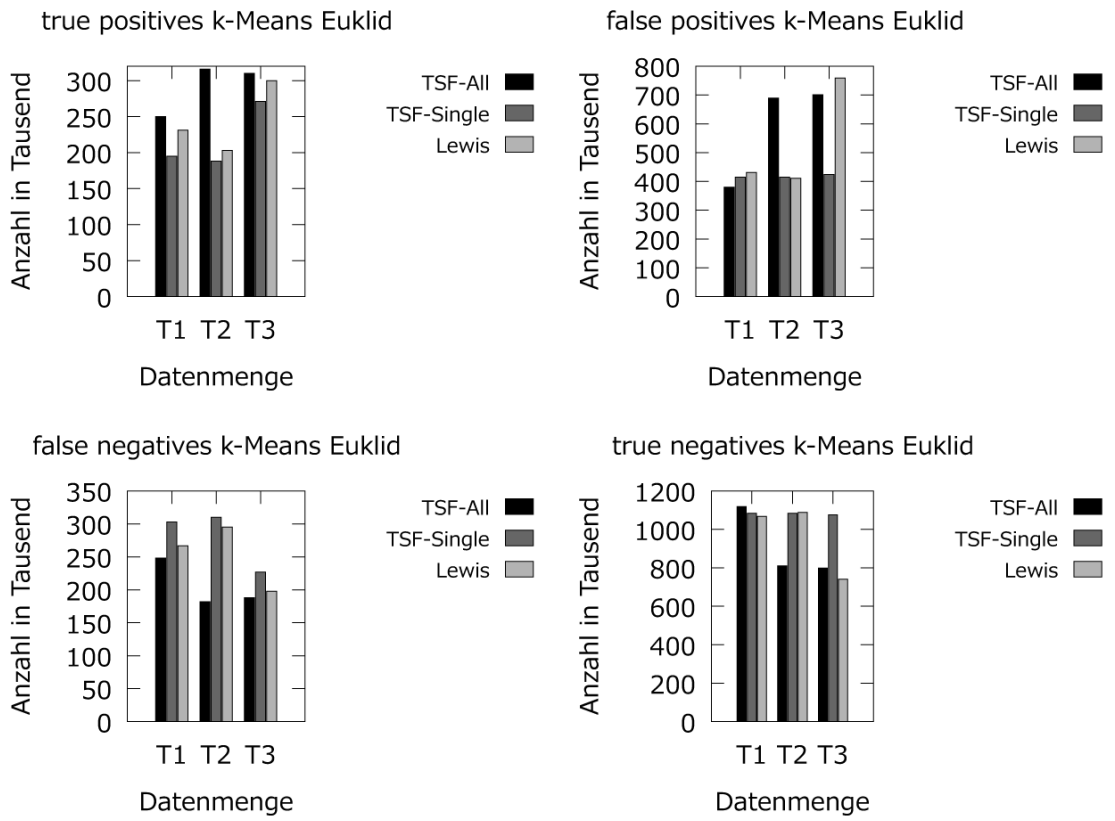


Abbildung 5.24: Evaluationswerte für den k-Means-Algorithmus mit euklidischem Distanzmaß

Stellt man die Anzahlen für den k-Means-Algorithmus mit Cosinus als Distanzmaß grafisch dar, so erhält man die folgenden Diagramme[1]:

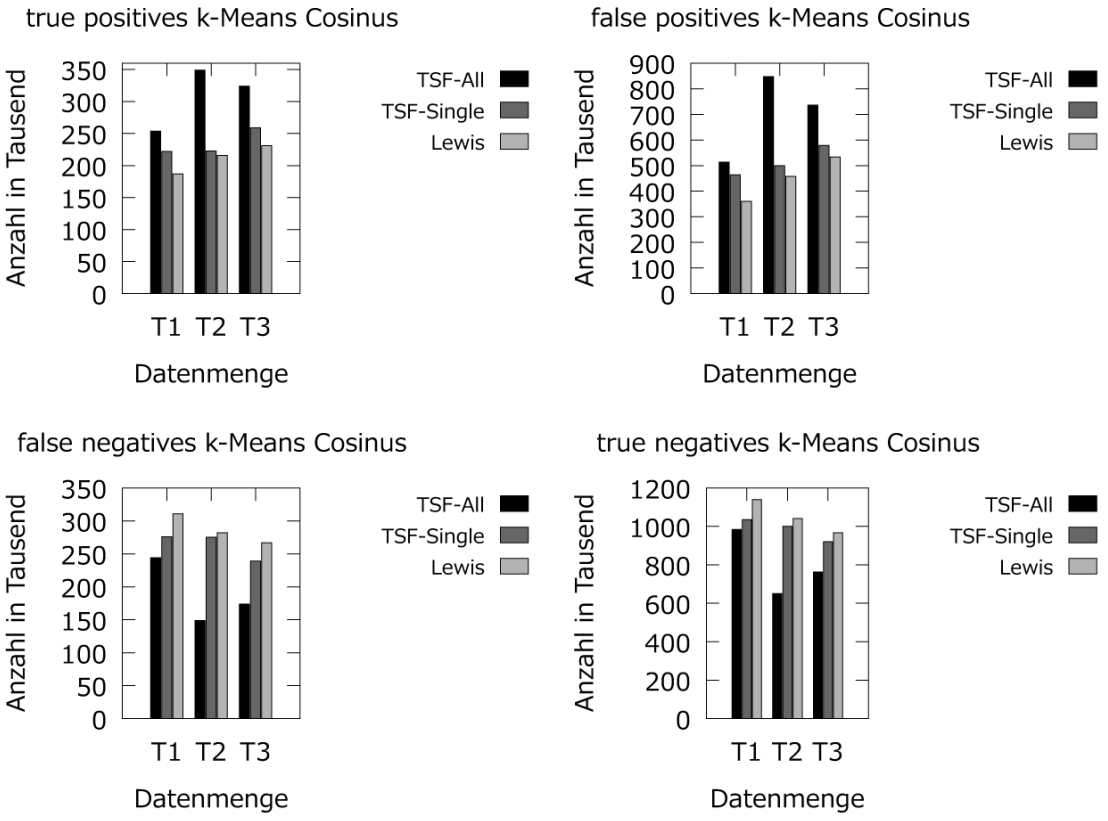


Abbildung 5.25: Evaluationswerte für den k-Means-Algorithmus mit Cosinus als Distanzmaß

Daran abzulesen ist:

1. Bei TP sind die Ergebnisse mit TSF-Feature-Vektoren besser als mit den einzelwortbasierten Feature-Vektoren. Die höchsten Anzahlen werden über alle drei Datenmengen durch die „AllDoks“-Feature-Vektoren erreicht. Darauf folgen die „SingleDoks“-Feature-Vektoren und die einzelwortbasierten Feature-Vektoren bilden den Abschluss. Das Ergebnis bei den TN ist ebenfalls eindeutig: Die einzelwortbasierten Feature-Vektoren erreichen hier die höchsten Anzahlen, darauf folgen die „SingleDoks“-Feature-Vektoren und zum Schluss die „AllDoks“-Feature-Vektoren.

1 Es werden die Werte betrachtet, die zum besten Durchlauf für die jeweilige Datenmenge geführt haben. Bei einem Gleichstand zwischen zwei Durchläufen wird ein Durchlauf zufällig ausgewählt.

2. Genau die gleiche Reihenfolge wie bei den TP ergibt sich für die FP.
3. Die FN haben die gleiche Reihenfolge wie die TN.

Stellt man die Werte für den GAAC-Algorithmus grafisch dar, so erhält man die folgenden Diagramme:

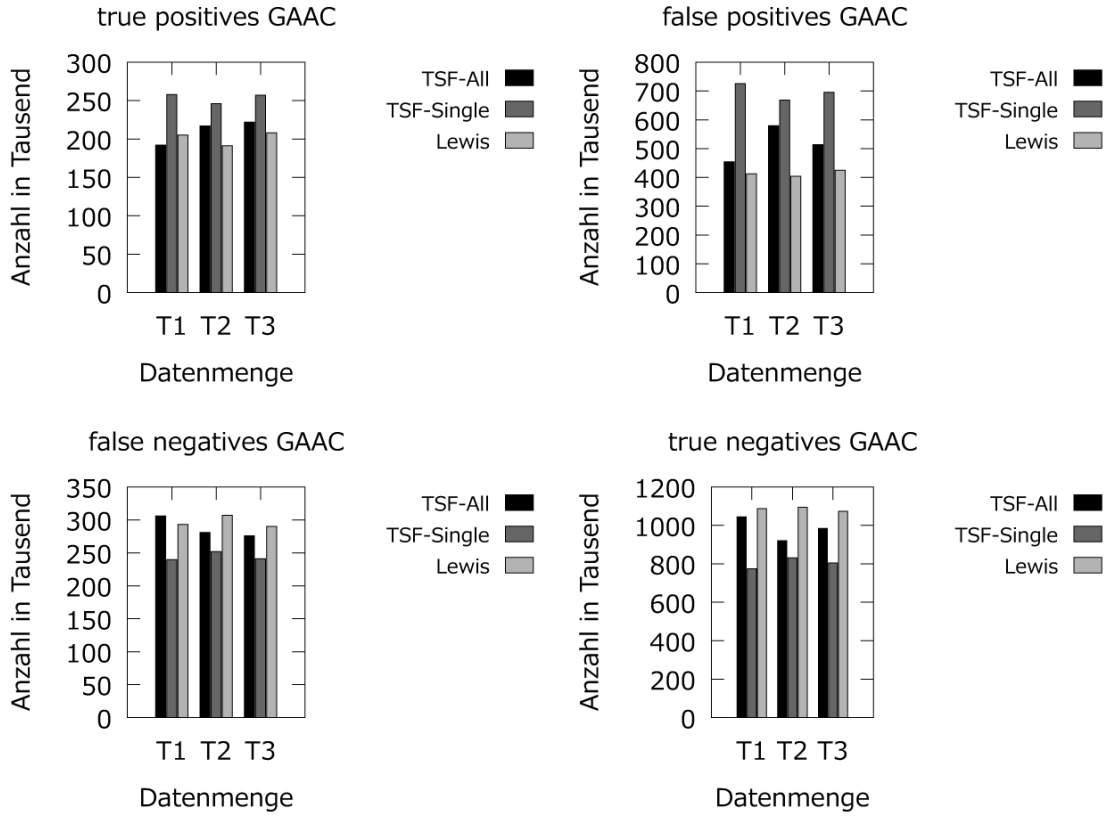


Abbildung 5.26: Evaluationswerte für den GAAC-Algorithmus

Daran abzulesen ist:

1. Bei den TP sind die Ergebnisse mit TSF-Feature-Vektoren besser als mit den einzelwortbasierten Feature-Vektoren. Die besten Werte werden über alle drei Datenmengen durch die „SingleDoks“-Feature-Vektoren erreicht. Darauf folgen die „AllDoks“-Feature-Vektoren und die einzelwortbasierten Feature-Vektoren bilden den Abschluss. Das Ergebnis bei den TN ist ebenfalls eindeutig: Die einzelwortbasierten Feature-Vektoren erreichen hier die höchsten Anzahlen, darauf folgen die „AllDoks“-Feature-Vektoren und zum Schluss die „SingleDoks“-Feature-Vektoren.
2. Genau die gleiche Reihenfolge wie bei den TP ergibt sich für die FP.

3. Die FN haben ebenfalls die gleiche Reihenfolge wie die TN.

Bildet man nun insgesamt eine Reihenfolge über alle Algorithmen und Distanzmaße für diese Werte, so ergibt sich Folgendes[1]:

- TP: „AllDoks“-Feature-Vektoren, „SingleDoks“-Feature-Vektoren, einzelwortbasierte Feature-Vektoren
- TN: einzelwortbasierte Feature-Vektoren, „SingleDoks“-Feature-Vektoren, „AllDoks“-Feature-Vektoren
- FP: „AllDoks“-Feature-Vektoren, „SingleDoks“-Feature-Vektoren, einzelwortbasierte Feature-Vektoren
- FN: einzelwortbasierte Feature-Vektoren, „SingleDoks“-Feature-Vektoren, „AllDoks“-Feature-Vektoren

Aus diesen Reihenfolgen lässt sich Folgendes schließen:

1. Die TP und FP sowie die TN und FN scheinen gleichermaßen zu steigen oder zu sinken. Das bedeutet, wenn die Anzahl der korrekt getroffenen Entscheidungen (TP und TN) durch die Cluster-Algorithmen steigt, dann steigt auch die Anzahl der entsprechenden Fehlentscheidungen (FP bzw. FN) an.
2. Auch durch die Betrachtung der den Evaluationsmaßen zu Grunde liegenden Werte lässt sich keine deutliche Überlegenheit eines der verwendeten Verfahren feststellen. Es herrscht ein Gleichstand zwischen den Ergebnissen des einzelwortbasierten Verfahrens zur Feature-Erzeugung und des wortübergreifenden Verfahrens zur Feature-Erzeugung.

   Im praktischen Kontext muss man in einem solchen Fall entscheiden, ob eher auf eine hohe Genauigkeit oder auf eine hohe Trefferquote Wert gelegt wird. Im erstgenannten Fall würde bspw. das $F_{0,5}$-Maß als Evaluationsmaß verwendet werden und man würde eher das einzelwortbasierte Verfahren verwenden. Im letztgenannten Fall würde bspw. das F_2-Maß als Evaluationsmaß verwendet werden und man würde die wortübergreifenden Verfahren präferieren.[2]

1 Dabei wird die Reihenfolge gebildet, indem die Mehrheit der erreichten Ergebnisse für die jeweilige Anzahl gezählt wird. Erreichen also die „AllDoks“-Feature-Vektoren mindestens zweimal bei TP die höchste Anzahl so erreichen sie insgesamt die höchste Anzahl, da das dann die Mehrheit aus den drei Datenmengen bildet.

2 Manning u.a. (2008), S. 331, schreiben, dass es in manchen Fällen wünschenswerter ist, unähnliche Dokumente dem gleichen Cluster zuzuordnen, anstatt ähnliche Dokumente zu trennen. Zusätzlich meinen die Autoren, dass Benutzer, solange sie nützliche Informationen erhalten, eine gewisse Toleranz für FP haben, vgl. Manning u.a. (2008), S. 143.

- Ist die Entscheidung, welches Verfahren gewählt werden soll, auch vom zu verwendenden Algorithmus abhängig, so ergibt sich für die durchgeführten Experimente, dass bei Verwendung des GAAC-Algorithmus ein einzelwortbasiertes Verfahren gewählt werden sollte, da in den Experimenten in 8 von 10 Fällen bessere Ergebnisse bei den Evaluationsmaßen erzielt werden als bei den wortübergreifenden Feature-Vektoren.[1] Das gilt jedoch nicht, wenn auf eine höhere Trefferquote Wert gelegt wird. Dann sollte eher ein wortübergreifendes Verfahren verwendet werden.

  Beim k-Means-Algorithmus mit beiden in den Experimenten verwendeten Distanzmaßen ist das Ergebnis nicht so deutlich wie beim GAAC-Algorithmus, jedoch erreichen die „AllDoks“-Feature-Vektoren in 9 von 20 Fällen ein besseres Ergebnis, während das für die einzelwortbasierten Feature-Vektoren nur in 6 von 20 Fällen der Fall ist. Beim k-Means-Algorithmus muss aber berücksichtigt werden, dass die Ergebnisse sehr stark von der zufälligen Wahl der ersten Clustermittelpunkte beeinflusst werden. Um eine wirklich sichere Aussage über die Verwendung von einzelwortbasierten oder wortübergreifenden Features treffen zu können, müsste in einer weiteren Arbeit diese Unsicherheit beseitig werden, indem die Clustermittelpunkte im ersten Schritt geeignet[2] ausgewählt und mit diesen die Experimente wiederholt werden.

Insgesamt lässt sich also keine deutliche Überlegenheit des wortübergreifenden oder des einzelwortbasierten Verfahrens beim Clustern von natürlichsprachlichen Dokumenten feststellen. Die Vermutung war, dass wortübergreifende Features bessere Ergebnisse liefern als einzelwortbasierte.
Ein möglicher Grund, warum die durchgeführten Experimente das nicht zeigen konnten, ist der folgende:
Die in dieser Arbeit durchgeführte Art der Feature-Erzeugung für die wortübergreifenden Features ist eventuell nicht für das Clustern der Dokumente geeignet.
Durch die Definition der TSF-Features müssen sie mindestens doppelt in einer bestimmten Menge von Dokumenten vorkommen. In den durchgeführten Experimenten wurden zwei Möglichkeiten, diese Definition zu erfüllen, realisiert: 1. Ein TSF-Feature muss mindestens doppelt in der Gesamtmenge der zu clusternden Dokumente vorkommen oder 2. ein TSF-Feature muss mindestens doppelt in einem der zu clusternden Dokumente vorkommen. Dadurch wird die Menge der potenziellen Features eingeschränkt, da so nicht jedes Text-Suffix-Fragment die Möglichkeit hat,

1 Die 10 Fälle entstehen, wenn „AllDoks“-Feature-Vektoren und „SingleDoks“-Feature-Vektoren nicht zusammengefasst betrachtet werden.
2 Siehe Kapitel 2.4.3.1 der vorliegenden Arbeit.

auch ein TSF-Feature zu werden. Die einzelwortbasierten Features decken dagegen alle vorkommenden Worte ab, da bei der Erzeugung dieser Features keine solche Einschränkung vorgenommen wurde. Im Fall der TSF-Features kann es also sein, dass Text-Suffix-Fragmente, die zu einem besseren Ergebnis beim Clustern geführt hätten, aufgrund der vorgenommenen Definition der TSF-Features nicht zu TSF-Features gemacht wurden.

6 Fazit

In der vorliegenden Arbeit konnte eine Lösung des wissenschaftlichen Problems, ob wortübergreifende Features bessere Klassifizierungs- und Clusterergebnisse als einzelwortbasierte Features bei der Klassifizierung und dem Clustern von natürlichsprachlichen Texten erreichen, präsentiert werden.
So wurde gezeigt, dass mit den TSF-Features, wie sie in der vorliegenden Arbeit definiert sind, das Klassifizieren und das Clustern natürlichsprachlicher Texte durchgeführt werden kann. Im Fall der Klassifizierung von natürlichsprachlichen Texten konnte eine Überlegenheit der wortübergreifenden Features gegenüber den einzelwortbasierten Features festgestellt werden. Im Fall des Clusterns von natürlichsprachlichen Texten wurde zwar keine Überlegenheit gezeigt, jedoch eine Gleichwertigkeit.
Daraus ergeben sich die folgenden *Empfehlungen* für den Einsatz von TSF-Features für das Klassifizieren und Clustern natürlichsprachlicher Texte:

- Beim *Klassifizieren* natürlichsprachlicher Texte bieten TSF-Features die Möglichkeit, die Ergebnisqualität zu steigern, ohne Zusatzwissen über Stoppworte, Stemming oder Featureselektion einfließen zu lassen. Die in der vorliegenden Arbeit durchgeführten Experimente mit einer Repräsentation der Texte, die mit Hilfe einer Datenstruktur namens Suffix Array erzeugt wurde, zeigen überlegene Ergebnisse beim Klassifizieren der Texte, ohne weitere Verbesserungen vornehmen zu müssen. Insbesondere die Kombination mit einem Support-Vector-Machine-Algorithmus, dem State-of-the-art-Algorithmus beim Klassifizieren, zeigt diese Überlegenheit deutlich. Aus diesem Grund wird der Einsatz von TSF-Features für das Klassifizieren von natürlichsprachlichen Texten von der Verfasserin empfohlen.

- Beim *Clustern* natürlichsprachlicher Texte empfiehlt die Verfasserin TSF-Features nur, wenn weitere Vorteile aus dem Einsatz der TSF-Features gezogen werden können. Wie in den Cluster-Experimenten gezeigt, sind TSF-Features und einzelwortbasierte Features im Hinblick auf das Clustern natürlichsprachlicher Texte gleichwertig. Ebenfalls wurde gezeigt, dass eine Überlegenheit von

TSF-Features nur unter bestimmten Voraussetzungen vorhanden ist[1]. Daher ist der Einsatz von TSF-Features beim Clustern natürlichsprachlicher Texte abhängig von dem Ziel, das beim Clustern erreicht werden soll.

Durch die Verwendung nur eines Testdatensatzes im Benchmark kann nicht von einer Allgemeingültigkeit der Ergebnisse gesprochen werden. Jedoch ist die Verfasserin der Meinung, dass durch das Einfließen von so wenig Zusatzwissen wie möglich die Ergebnisse auch mit anderen Texten erreicht werden können. Die Repräsentation der Texte wurde durch eine Aufbereitung dieser Texte mit Hilfe der Datenstruktur Suffix Array erzeugt. Bei dieser Aufbereitung ist kein Wissen über Wortgrenzen, grammatikalische Strukturen oder Inhalte beachtet worden, es wurde lediglich eine „Umformung“ der Texte durchgeführt. Diese Umformung lässt sich auf jeden anderen natürlichsprachlichen Text, auch in einer anderen Sprache, die auf lateinischen Buchstaben[2] basiert, anwenden. Somit ist die Erzeugung der TSF-Features auch auf andere Texte übertragbar.
Da im Verlauf der Experimente keine Änderung der Repräsentation der Texte durch Stemming, Stoppwortentfernung oder eine andere Featureselektion als die, die sich durch die Definition der TSF-Features ergibt, durchgeführt wurde, sondern nur die verwendeten Algorithmen mit „Basisparametern“[3] auf die Textrepräsentation angewendet wurden, ist zu erwarten, dass die erzeugten Ergebnisse, also die Gleichwertigkeit oder Überlegenheit der neuartigen Features beim Clustern und Klassifizieren, auf andere Datensätze übertragbar sind.
Diese Übertragbarkeit ist auch ein Ansatz für die Lösung des Realproblems. Das automatische Erstellen von Klassifikationen bleibt schwierig, da die Experimente keine Qualitätssteigerung bei der Verwendung von TSF-Features im Vergleich zu einzelwortbasierten Features beim Clustern ergeben haben. Jedoch wurden alle Ergebnisse ohne Zusatzwissen erzeugt und bei bereits vorhandenen Klassifikationen wurde eine Steigerung der Qualität bei der „Einsortierung“ neuer Dokumente erreicht.
Die betriebswirtschaftlich wünschenswerte Situation ist, auf das Beispiel der Bearbeitung unwichtiger E-Mails aus Kapitel 1 bezogen, dass aus den bereits vorhandenen E-Mails automatisiert ein zu 100% korrektes Clustering erzeugt wird. Damit

1 In den Experimenten war das vor allem die Voraussetzung, dass auf eine höhere Trefferquote Wert gelegt wird.

2 Die Umformung sprachspezifischer Buchstaben, wie die deutschen Umlaute, muss vorher vorgenommen werden oder ihre Behandlung in den Algorithmus zur Erzeugung des Suffix Arrays einfließen.

3 Damit ist gemeint, dass auch im Fall der verwendeten Algorithmen in den Experimenten eine möglichst einfache Form dieser Algorithmen ohne Verbesserung bestimmter Parameter (wie Begrenzung der Featureanzahl oder Auswahl eines problemspezifisch günstigen Kernels) gewählt wurde.

ist eine Klassifikation vorhanden, auf deren Grundlage neu hinzukommende E-Mails automatisiert klassifiziert werden können. Geschieht dies ebenfalls völlig korrekt, also zu 100% richtig, entstehen dem Unternehmen keine Kosten mehr durch die Bearbeitung unwichtiger E-Mails.
Durch den in der vorliegenden Arbeit erstellten Software-Prototyp[1] lässt sich diese wünschenswerte Situation nicht realisieren, aber durch den Einsatz des Software-Prototyps könnten bereits Kosten eingespart werden.
Nimmt man an, dass die Übertragbarkeit der Ergebnisse der Experimente der vorliegenden Arbeit auch auf andere natürlichsprachliche Texte gegeben ist und bereits vorhandene E-Mails zu 100% richtig in eine Klassifikation einsortiert wurden[2], so ergibt sich folgende grobe Abschätzung der Kostenersparnis:

- Ein Mitarbeiter erhält neue E-Mails, die aufgrund der nicht-hierarchischen Klassifikation der E-Mails des Mitarbeiters automatisch klassifiziert werden.
  - In diesem Szenario erreicht der 10-Nearest-Neighbour-Klassifizierer mit 72,75% richtig klassifizierter Dokumente die beste Qualität und der Naive-Bayes-Klassifizierer mit 36,95% richtig klassifizierter Dokumente die schlechteste Qualität.[3]
  - Das bedeutet, im besten Fall werden ungefähr 73% der neu hinzukommenden E-Mails automatisiert in die richtigen Klassen eingeordnet. Übrig bleiben ca. 27% der E-Mails, die der Mitarbeiter manuell bearbeiten muss, d.h., er muss diese E-Mails bearbeiten, ohne zu wissen, ob sie wichtig oder unwichtig für ihn sind. Der Anteil der unwichtigen E-Mails am Gesamtumfang neu hinzukommender E-Mails liegt laut Zeldes u.a. (2007)[4] bei 30%. Bei insgesamt 350 Mails, die jeder Mitarbeiter pro Woche erhält[5], sind also 105 E-Mails unwichtig. Sind nun bereits 73% dieser unwichtigen E-Mails richtig klassifiziert, müssen nur noch 28,35 unwichtige E-Mails

1 Siehe dazu Anhang F auf S. 593.
2 In diesem Szenario wird die Klassifizierung neu hinzukommender E-Mails betrachtet, daher wird in der Ausgangssituation davon ausgegangen, dass eine Klassifikation besteht. Diese kann manuell oder automatisiert erstellt worden sein. Insbesondere wird aus den in der Klassifikation vorhandenen Klassen ersichtlich, welche Klassen für den jeweiligen Mitarbeiter wichtige und unwichtige E-Mails enthalten.
3 Das bessere Ergebnis wird im Experiment Klass1RV für die Trainingsmenge Tr2 und die Testmenge Te5 erreicht. Das schlechtere Ergebnis stammt aus dem Experiment Klass1RV für die Trainingsmenge Tr2 und die Testmenge Te2. Das Experiment Klass1RV wurde für das Szenario ausgewählt, da es sich auf die Klassenfamilie Region bezieht, für die keine Hierarchie vorhanden ist.
4 Vgl. Zeldes u.a. (2007), S. 6 (Seitennummerierung durch die Verfasserin, da Webdokument).
5 Vgl. Zeldes u.a. (2007), S. 6 (Seitennummerierung durch die Verfasserin, da Webdokument).

bearbeitet werden.[1] Die Arbeitszeit, die jeder Mitarbeiter benötigt, um seine unwichtigen E-Mails zu bearbeiten, wird also von 2 Stunden[2] pro Woche für 105 E-Mails auf 32,4 Minuten, also 0,54 Stunden pro Woche reduziert. Daraus ergeben sich bei ansonsten unveränderten Werten für das Unternehmen:

$$\begin{aligned}\text{Kosten des Unternehmens für die Bearbeitung unwichtiger E-Mails} &= \frac{0,54 \text{ Stunden}}{\text{Woche und Mitarbeiter}} * \frac{50 \$}{\text{Stunde}} * \\ &\quad \frac{49 \text{ Wochen}}{\text{Jahr}} * 50.000 \text{ Mitarbeiter} \\ &= \frac{66.150.000 \$}{\text{Jahr}}\end{aligned}$$

Bei zuvor angesetzten Kosten von 245.000.000 $ pro Jahr für die Bearbeitung von unwichtigen E-Mails würde mit einer zu 73% richtigen Klassifizierung unter den zuvor genannten Voraussetzungen im besten Fall eine Kostenersparnis für das Unternehmen von 178.850.000 $ pro Jahr erreicht werden.

– Im schlechtesten Fall werden ungefähr 37% der neu hinzukommenden E-Mails automatisiert in die richtigen Klassen eingeordnet. Übrig bleiben ca. 63% der E-Mails, die der Mitarbeiter manuell bearbeiten muss, d.h., er muss diese E-Mails bearbeiten, ohne zu wissen, ob sie wichtig oder unwichtig für ihn sind. Der Anteil der unwichtigen E-Mails am Gesamtumfang neu hinzukommender E-Mails liegt laut Zeldes u.a. (2007)[3] bei 30%. Bei insgesamt 350 Mails, die jeder Mitarbeiter pro Woche erhält[4], sind also 105 E-Mails unwichtig. Sind nun 37% dieser unwichtigen E-Mails richtig klassifiziert, müssen noch 66,15 unwichtige E-Mails bearbeitet werden. Die Arbeitszeit, die jeder Mitarbeiter benötigt, um seine unwichtigen E-Mails zu bearbeiten, wird also von 2 Stunden[5] pro Woche für 105 E-Mails auf 75,6 Minuten, also 1,26 Stunden pro Woche reduziert.

1 In der vorliegenden Arbeit werden alle Dokumente klassifiziert. Das würde bedeuten, dass sich diese 28,35 E-Mails in den falschen Klassen befänden und zunächst ermittelt werden müssten. Dies wäre mit Kosten verbunden. Also wird in diesem und den folgenden Szenarien davon ausgegangen, dass sich diese unwichtigen und nicht-korrekt klassifizierten E-Mails im Posteingang befinden, d.h., falsch klassifizierte E-Mails werden in den Posteingang „einsortiert“.

2 Vgl. Zeldes u.a. (2007), S. 6 (Seitennummerierung durch die Verfasserin, da Webdokument).

3 Vgl. Zeldes u.a. (2007), S. 6 (Seitennummerierung durch die Verfasserin, da Webdokument).

4 Vgl. Zeldes u.a. (2007), S. 6 (Seitennummerierung durch die Verfasserin, da Webdokument).

5 Vgl. Zeldes u.a. (2007), S. 6 (Seitennummerierung durch die Verfasserin, da Webdokument).

Daraus ergeben sich bei ansonsten unveränderten Werten für das Unternehmen:

$$\begin{aligned}\text{Kosten des Unternehmens für die Bearbeitung unwichtiger E-Mails} &= \frac{1,26 \text{ Stunden}}{\text{Woche und Mitarbeiter}} * \frac{50 \text{ \$}}{\text{Stunde}} * \\ &\quad \frac{49 \text{ Wochen}}{\text{Jahr}} * 50.000 \text{ Mitarbeiter} \\ &= \frac{154.350.000 \text{ \$}}{\text{Jahr}}\end{aligned}$$

Bei zuvor angesetzten Kosten von 245.000.000 $ pro Jahr für die Bearbeitung von unwichtigen E-Mails würde mit einer zu 37% richtigen Klassifizierung unter den zuvor genannten Voraussetzungen im schlechtesten Fall immer noch eine Kostenersparnis für das Unternehmen von 90.650.000 $ pro Jahr erreicht werden.

- Ein Mitarbeiter erhält neue E-Mails, die aufgrund der hierarchischen Klassifikation der E-Mails des Mitarbeiters automatisch klassifiziert werden.
  - In diesem Szenario erreicht der Support-Vector-Machine-Klassifizierer mit 94,10% richtig klassifizierter Dokumente die beste Qualität und der Decision-Tree-Klassifizierer mit 11,31% richtig klassifizierter Dokumente die schlechteste Qualität.[1]
  - Das bedeutet, im besten Fall werden ungefähr 94% der neu hinzukommenden E-Mails automatisiert in die richtigen Klassen eingeordnet.[2] Übrig

1 Das bessere Ergebnis wird im Experiment Klass1TV für die Trainingsmenge Tr1 und die Testmenge Te6 erreicht. Der Support-Vector-Machine-Klassifizierer erreicht in diesem Experiment ein vergleichbares Ergebnis zum 10-Nearest-Neighbour-Klassifizierer. Der Support-Vector-Machine-Klassifizierer wurde als bester Klassifizierer in diesem Szenario ausgewählt, damit alle Klassifizierer in den in diesem Kapitel genannten Szenarien berücksichtigt werden. Das schlechteste Ergebnis stammt aus dem Experiment Klass1IV für die Trainingsmenge Tr1 und die Testmenge Te7. Die Experimente Klass1TV und Klass1IV wurden für das Szenario ausgewählt, da sie sich auf die Klassenfamilien Topic und Industry beziehen, für die Hierarchien vorhanden sind.

2 Dabei ist zu beachten, dass folgende Einschränkung gilt: Die in den Experimenten gemessene Qualität der Klassifizierer stellt eine korrekte Klassifizierung eines Dokuments dann fest, wenn dieses einer korrekten Klasse innerhalb des „Hierarchiezweigs“ zugeordnet wurde. Das bedeutet, es wird die korrekte Klasse zugeordnet, jedoch kann es eine speziellere Klasse, also eine Klasse in einer tieferen Hierarchieebene, geben, die ebenfalls korrekt gewesen wäre. Eine speziellere Einordnung der E-Mails müsste dann manuell, mit weiteren Kosten verbunden, vorgenommen werden. In der erfolgten groben Abschätzung wird davon ausgegangen, dass für einen Mitarbeiter Klassen mit für ihn unwichtigen E-Mails in der Hierarchie keine Spezialisierungen oder Generalisierungen von Klassen mit für ihn wichtigen E-Mails sind.

bleiben ca. 6% der E-Mails, die der Mitarbeiter manuell bearbeiten muss, d.h., er muss diese E-Mails bearbeiten, ohne zu wissen, ob sie wichtig oder unwichtig für ihn sind. Der Anteil der unwichtigen E-Mails am Gesamtumfang neu hinzukommender E-Mails liegt laut Zeldes u.a. (2007)[1] bei 30%. Bei insgesamt 350 Mails, die jeder Mitarbeiter pro Woche erhält[2], sind also 105 E-Mails unwichtig. Sind nun bereits 94% dieser unwichtigen E-Mails richtig klassifiziert, müssen nur noch 6,3 unwichtige E-Mails bearbeitet werden. Die Arbeitszeit, die jeder Mitarbeiter benötigt, um seine unwichtigen E-Mails zu bearbeiten, wird also von 2 Stunden[3] pro Woche für 105 E-Mails auf 7,2 Minuten, also 0,12 Stunden pro Woche reduziert. Daraus ergeben sich bei ansonsten unveränderten Werten für das Unternehmen:

$$\begin{aligned}\text{Kosten des Unternehmens für die Bearbeitung unwichtiger E-Mails} &= \frac{0,12 \text{ Stunden}}{\text{Woche und Mitarbeiter}} * \frac{50 \text{ \$}}{\text{Stunde}} * \\ &\quad \frac{49 \text{ Wochen}}{\text{Jahr}} * 50.000 \text{ Mitarbeiter} \\ &= \frac{14.700.000 \text{ \$}}{\text{Jahr}}\end{aligned}$$

Bei zuvor angesetzten Kosten von 245.000.000 $ pro Jahr für die Bearbeitung von unwichtigen E-Mails würde mit einer zu 94% richtigen Klassifizierung unter den zuvor genannten Voraussetzungen im besten Fall eine Kostenersparnis für das Unternehmen von 230.300.000 $ pro Jahr erreicht werden.

– Im schlechtesten Fall werden ungefähr 11% der neu hinzukommenden E-Mails automatisiert in die richtigen Klassen eingeordnet. Übrig bleiben ca. 89% der E-Mails, die der Mitarbeiter manuell bearbeiten muss, d.h., er muss diese E-Mails bearbeiten, ohne zu wissen, ob sie wichtig oder unwichtig für ihn sind. Der Anteil der unwichtigen E-Mails am Gesamtumfang neu hinzukommender E-Mails liegt laut Zeldes u.a. (2007)[4] bei 30%. Bei insgesamt 350 Mails, die jeder Mitarbeiter pro Woche erhält[5],

1 Vgl. Zeldes u.a. (2007), S. 6 (Seitennummerierung durch die Verfasserin, da Webdokument).
2 Vgl. Zeldes u.a. (2007), S. 6 (Seitennummerierung durch die Verfasserin, da Webdokument).
3 Vgl. Zeldes u.a. (2007), S. 6 (Seitennummerierung durch die Verfasserin, da Webdokument).
4 Vgl. Zeldes u.a. (2007), S. 6 (Seitennummerierung durch die Verfasserin, da Webdokument).
5 Vgl. Zeldes u.a. (2007), S. 6 (Seitennummerierung durch die Verfasserin, da Webdokument).

sind also 105 E-Mails unwichtig. Sind nun 11% dieser unwichtigen E-Mails richtig klassifiziert, müssen noch 93,45 unwichtige E-Mails bearbeitet werden. Die Arbeitszeit, die jeder Mitarbeiter benötigt, um seine unwichtigen E-Mails zu bearbeiten, wird also von 2 Stunden[1] pro Woche für 105 E-Mails auf 106,8 Minuten, also 1,78 Stunden pro Woche reduziert. Daraus ergeben sich bei ansonsten unveränderten Werten für das Unternehmen:

$$\begin{aligned}\text{Kosten des Unternehmens für die Bearbeitung unwichtiger E-Mails} &= \frac{1,78 \text{ Stunden}}{\text{Woche und Mitarbeiter}} * \frac{50 \$}{\text{Stunde}} * \\ &\quad \frac{49 \text{ Wochen}}{\text{Jahr}} * 50.000 \text{ Mitarbeiter} \\ &= \frac{218.050.000 \$}{\text{Jahr}}\end{aligned}$$

Bei zuvor angesetzten Kosten von 245.000.000 $ pro Jahr für die Bearbeitung von unwichtigen E-Mails würde mit einer zu 11% richtigen Klassifizierung unter den zuvor genannten Voraussetzungen im schlechtesten Fall immer noch eine Kostenersparnis für das Unternehmen von 26.950.000 $ pro Jahr erreicht werden.

Nimmt man die Übertragbarkeit der Ergebnisse der Experimente der vorliegenden Arbeit auf andere natürlichsprachliche Texte an und will man aufgrund der vorhandenen E-Mails eine Klassifikation erst erstellen, also die vorhandenen E-Mails clustern, so lässt sich keine grobe Abschätzung für die Kostenersparnis erstellen. Das liegt daran, dass im Beispiel aus Kapitel 1 keine Erhebung vorhanden ist, die die Kosten für eine Erzeugung einer Klassifikation enthält. Bei der Bearbeitung unwichtiger E-Mails werden nur Kosten genannt, die auftreten, wenn Dokumente neu hinzukommen. Es können daher nur folgende Aussagen getroffen werden:

- Das bestmögliche Ergebnis in den Experimenten der vorliegenden Arbeit wurde im Experiment Clus2RV für den k-Means-Clusterer mit euklidischem Distanzmaß für die TSF-Feature-Erzeugungsart „AllDoks“ im ersten Durchlauf für die Datenmenge T1 erreicht. In diesem Experiment wurde eine Qualität des erzeugten Clusterings von 68,52% richtig geclusterter Dokumente durch den Rand Index gemessen. Das würde, übertragen auf das Beispiel eines automatischen Clusterns der vorhandenen E-Mails, bedeuten, dass ein Clustering

1 Vgl. Zeldes u.a. (2007), S. 6 (Seitennummerierung durch die Verfasserin, da Webdokument).

erstellt wird, in dem der angegebene Pozentsatz der Dokumente laut Rand Index korrekt geclustert worden wäre. Die restlichen Dokumente müssten manuell entweder in ihr korrektes Cluster einsortiert werden oder es müssten noch weitere Cluster manuell erzeugt werden und die restlichen Dokumente dort einsortiert werden.[1] Mögliche Kosten für das manuelle Erstellen einer Klassifikation könnten also unter den bereits zur Klassifizierung und zum Clustern erläuterten Einschränkungen um ungefähr 69% gesenkt werden.

- Das schlechteste Ergebnis in den Experimenten der vorliegenden Arbeit wurde im Experiment Clus4RV für den k-Means-Clusterer mit Cosinus als Distanzmaß für die TSF-Feature-Erzeugungsart „SingleDoks“ im ersten Durchlauf für die Datenmenge T3 erreicht. In diesem Experiment wurde eine Qualität des erzeugten Clusterings von 29,06% richtig geclusterter Dokumente durch das $F_{0,5}$-Maß gemessen. Das würde, übertragen auf das Beispiel eines automatischen Clusterns der vorhandenen E-Mails, bedeuten, dass ein Clustering erstellt wird, in dem der angegebene Pozentsatz der Dokumente laut $F_{0,5}$-Maß korrekt geclustert worden wäre. Die restlichen Dokumente müssten manuell entweder in ihr korrektes Cluster einsortiert werden oder es müssten noch weitere Cluster manuell erzeugt werden und die restlichen Dokumente dort einsortiert werden. Mögliche Kosten für das manuelle Erstellen einer Klassifikation könnten also unter den bereits zur Klassifizierung und zum Clustern erläuterten Einschränkungen um ungefähr 29% gesenkt werden.

Somit könnten - mit allen genannten Einschränkungen - Kosten in Unternehmen bereits bei Einsatz des vorgestellten Software-Prototyps bei der Bearbeitung unwichtiger E-Mails eingespart werden.
Neben der Einschränkung der eventuellen Nicht-Allgemeingültigkeit der Ergebnisse existiert noch eine weitere Einschränkung des hier gewählten Ansatzes: Die Betrachtung der Dokumente und demzufolge auch die TSF-Feature-Erzeugung erfolgt über die syntaktische Ebene der Texte. Es werden also keine semantischen Betrachtungen einbezogen. Diese Einschränkung wurde von der Verfasserin bereits in der wissenschaftlichen Problemstellung vorgenommen, da sie eine Qualitätsverbesserung ohne explizites Zusatzwissen anstrebte. Semantische Betrachtungen und damit der Einsatz von Ontologien, die ebenfalls erstellt und gewartet werden müssen, erfordern ein Zusatzwissen. Ontologien müssen an das jeweilige Unternehmen angepasst werden, um bspw. für das Klassifizieren neu hinzukommender E-Mails verwendet werden zu

1 Da in den in der vorliegenden Arbeit durchgeführten Experimenten immer alle Dokumente geclustert wurden, müssten in einem solchen Szenario zunächst die falsch geclusterten Dokumente bestimmt und ihre Zuordnung korrigiert werden.

können. D.h., ein Software-Prototyp, der die Semantik in den E-Mails berücksichtigen würde, müsste von Experten an das jeweilige Unternehmen angepasst werden. Hinzu kommt noch, dass die Verfasserin anstrebte, bei der Klassifizierung und dem Clustern von Dokumenten nur deren wirklichen Inhalte[1] zu berücksichtigen. Die in den Dokumenten enthaltene Semantik wird zwar vom Menschen als wirklicher Inhalt der Dokumente wahrgenommen, jedoch nicht von einem Computer. Für den Computer besteht der wirkliche Inhalt aus Zeichensequenzen. Diese bildeten in der vorliegenden Arbeit die Grundlage für das Klassifizieren und das Clustern, um weiteres, ansonsten benötigtes Zusatzwissen zu vermeiden.

1 Für eine Erläuterung siehe Fußnote 1 auf S. 31.

7 Ausblick

Die vorliegende Arbeit führt neben wortbasierten Features und Multi-Word-Features eine dritte Featureart ein, die zur Klassifizierung und zum Clustern von natürlichsprachlichen Texten in Erwägung zu ziehen ist. So konnte gezeigt werden, dass sich wortübergreifende Features für das Klassifizieren natürlichsprachlicher Texte sehr gut eigenen und auch für das Clustern von natürlichsprachlichen Texten eingesetzt werden können. Jedoch konnte die Arbeit nicht alle Facetten dieses Themengebiets beleuchten und auch keine allgemeingültigen Ergebnisse liefern.
In weiteren Arbeiten zu diesem Thema müssten also vor allen Dingen weitere Benchmarks mit anderen Testdatensätzen durchgeführt werden, um die Übertragbarkeit der Ergebnisse der vorliegenden Arbeit zu prüfen. Bisher wurden nur Experimente mit einem Testdatensatz in englischer Sprache durchgeführt. In Bezug auf die Sprache ergeben sich weitere Möglichkeiten, Benchmarks durchzuführen. So sollten, unter Berücksichtigung von spracheigenen Spezifika, weitere Texte mit lateinischen Buchstaben ebenfalls als Benchmarkdatensätze für die Klassifizierung und das Clustern mit TSF-Features herangezogen werden.
Denkbar ist es jedoch auch, Texte in Sprachen ohne lateinische Buchstaben zu verwenden. Die Aufbereitung der Texte mit Hilfe von Suffix Arrays und die Erzeugung von Features aufgrund von gleichen Zeichen, die doppelt im Text oder in mehreren Texten vorkommen und mindestens drei Zeichen umfassen, ist unabhängig vom lateinischen Alphabet, da sie nur auf gleichen *Zeichen* basiert. So können auch Texte verarbeitet werden, die andere Zeichen in ihrer Sprache verwenden.[1] Die nötigen Anpassungen müssen dann in den Algorithmus zur Erzeugung der Suffix Arrays einfließen, nicht jedoch in die Art und Weise, wie die Features erzeugt werden. Das bedeutet, lediglich die Sortierung der Zeichen im Suffix Array unterscheidet sich voneinander. Das könnte in weiteren Arbeiten geprüft werden.
Bezogen auf das Klassifizieren von natürlichsprachlichen Texten, so wie hier in dieser Arbeit vorgenommen, ergibt sich weiterer Forschungsbedarf im Hinblick auf die weitere qualitative Verbesserung der Ergebnisse. Insbesondere die guten Ergebnisse der TSF-Features in Kombination mit der Support Vector Machine lassen viel Spielraum für weitere Arbeiten erkennen. Die Support Vector Machine selbst wurde in

1 Ein Beispiel dafür ist die Erstellung eines Suffix Arrays für Texte in japanischer Sprache, vgl. Yamamoto u.a. (2001), S. 17.

der vorliegenden Arbeit mit einem linearen Kernel verwendet. In weiteren Arbeiten könnten andere, nicht-lineare Kernel daraufhin überprüft werden, ob mit ihnen weitere Qualitätssteigerungen zu erreichen sind. Des Weiteren müsste geprüft werden, ob eine eventuelle weitergehende Featureselektion ebenfalls zu Qualitätssteigerungen beitragen kann. Dafür müssten Methoden für die TSF-Featureselektion entwickelt werden.[1] Dadurch kann auch der benötigte Rechenaufwand gesenkt werden. Eine weitere Möglichkeit wäre das Einbeziehen von Gewichten in den Feature-Vektoren. Das hieße, dass die TSF-Feature-Vektoren nicht nur das Vorhandensein eines TSF-Features im Text abbilden, sondern dieses Feature - bspw. mit tf-idf[2] - gewichtet dargestellt wird.[3]

In Bezug auf das Clustern von natürlichsprachlichen Texten ergibt sich weiterer Forschungsbedarf bei der Erzeugung von wortübergreifenden Features. Durch die Experimente konnte keine Überlegenheit von wortübergreifenden Features festgestellt werden. Eine Vermutung ist, dass die Definition der TSF-Features so, wie sie in dieser Arbeit aufgestellt wurde, ein möglicher Grund dafür ist. In weiteren Arbeiten könnten andere Möglichkeiten der wortübergreifenden Feature-Erzeugung erprobt werden.[4] Ein Vorteil, den der Einsatz von TSF-Features gegenüber wortbasierten Features beim Clustern haben könnte, wäre es, eine für den Menschen verstehbare Zusammenfassung des Clusterinhalts aufgrund der Features direkt erzeugen zu können, ohne weitere Techniken einsetzen zu müssen.[5]

Interessant wäre es zudem, einen Vergleich mit Multi-Wort-Features durchzuführen. Das könnte in weiteren Arbeiten durchgeführt werden.

1 Bisher werden alle TSF-Features, die der Definition entsprechen, für die Klassifizierung und das Clustern verwendet. Durch die Erzeugungsart dieser Features mit Hilfe der Datenstruktur Suffix Array sind jedoch per definitionem viele Features vorhanden, die in ihrer Anfangs-Zeichensequenz gleich sind. Aus dieser Tatsache könnten sich Methoden für die Selektion von Features ergeben.

2 Siehe S. 27 der vorliegenden Arbeit.

3 Eine Möglichkeit, die term frequency und die document frequency mit Suffix Arrays zu berechnen, findet sich in Yamamoto u.a. (2001), S. 13-15.

4 Eine Möglichkeit ist es, genau wie bei wortbasierten Features auf das doppelte Vorkommen der Features zu verzichten und alle Suffixe des gesamten Textes zu verwenden. Das ergibt aber die Schwierigkeit, dass der Rechen- und Speicheraufwand signifikant anwächst, es müsste also selektiert werden, siehe Fußnote 1 auf S. 546.

5 Stellt man sich einen durchgeführten Cluster-Vorgang vor, so werden Cluster durch einen Algorithmus aufgrund der Repräsentationen der Texte erzeugt. Nimmt man an, dass es sich um unbekannte Texte handelt, so sind diese danach in Cluster unterteilt, die jedoch nicht benannt sind. Der Mensch weiß also nur, dass eine gewisse Anzahl an Clustern existiert und welche Texte in ihnen enthalten sind, jedoch nicht, was das „Thema“ eines jeden Clusters ist. Um die Cluster zu benennen, kann auf die Features als Repräsentanten der enthaltenen Texte zurückgegriffen werden. Für den Menschen könnten dann zusammenhängende Textfragmente sinnvoller erscheinen als einzelne und gestemmte Worte. Ein Vorschlag, wie das mit einem Suffix Tree für ein erzeugtes Clustering durchgeführt werden kann, befindet sich in Chim u.a. (2007), S. 125.

Ein weiterer Forschungsansatz, der die beschriebenen Probleme lösen könnte, wäre der völlige Verzicht auf die Repräsentation der Texte durch Features und Vektoren. Das würde bedeuten, dass die Texte direkt auf ihre Ähnlichkeit hin verglichen werden müssten, ohne zuvor ihren Zusammenhang zu zerstören und die so verloren gegangenen Informationen bspw. durch Featureselektion und Parametereinstellung der Algorithmen zu „rekonstruieren“, so dass eine Klassifizierung und ein Clustern der Texte mit „guter“ Qualität erfolgen kann. Bisher, auch in der vorliegenden Arbeit, sind die erreichten Ergebnisse von der gewählten Repräsentation der Texte, dem gewählten Ähnlichkeitsmaß und dem gewählten Algorithmus mit seinen Parametereinstellungen abhängig. Entfällt eine Repräsentation der Texte, da sie sich selbst repräsentieren, entfällt auch die erste der drei genannten Abhängigkeiten.
Dieser Ansatz der featurelosen Klassifizierung und des featurelosen Clusterns würde bedeuten, dass neue Ähnlichkeitsmaße entwickelt werden müssen, die nicht auf einer Vektordarstellung der Texte beruhen, sondern auf den Texten selbst. Eine Möglichkeit in dieser Richtung wäre ein Ähnlichkeitsmaßstab, der auf Suffix Arrays oder Suffix Trees „arbeitet“, da diese Datenstrukturen einen Text aufbereiten, jedoch nicht direkt auf Features als Repräsentanten eingrenzen. Durch die Aufbereitung eines Textes als Suffix Array oder als Suffix Tree und das Hinzufügen eines zweiten Textes in die jeweilige Datenstruktur wird das ursprüngliche Suffix Array oder der ursprüngliche Suffix Tree in seiner „Form“ verändert. Wird diese „Formänderung“ messbar gemacht, kann daraus eine Ähnlichkeit der beiden Texte abgeleitet werden. Dieser Ansatz könnte weiter erforscht werden.[1]
Zwei Beispiele für Ähnlichkeitsmaße, die nicht auf Features basieren, sondern den Text und auch andere Daten direkt auf ihre Ähnlichkeit hin untersuchen, sind:

- Normalized Compression Distance (NCD)[2]
  Dieses Ähnlichkeitsmaß beruht auf Kompression. Die Texte werden einmal konkateniert komprimiert, davon der kleinere Wert, der sich aus der Kompression der einzelnen Texte ergibt, abgezogen und dieser Wert durch den größeren Wert, der sich aus der Kompression der einzelnen Texte ergibt, dividiert. So ergibt sich eine Ähnlichkeit zwischen den beiden Texten, denn je mehr Gemein-

1 Es existiert bereits ein Ähnlichkeitsmaß für Suffix Trees beim Clustern von Dokumenten, vgl. Chim u.a. (2007), S. 123 f. Die Autoren verwenden einen Suffix-Tree-Clustering-Algorithmus, der von Zamir u.a. (1998), S. 46-54, entwickelt wurde. Sie ergänzen den Algorithmus jedoch um ein Ähnlichkeitsmaß, das direkt auf dem Suffix Tree arbeitet. Allerdings wird der Suffix Tree selbst nicht wie in der vorliegenden Arbeit zeichenbasiert aufgebaut, sondern einzelwortbasiert. Es werden also keine wortübergreifenden Features verwendet. Außerdem werden zwar zunächst mit einem Suffix-Tree-basierten Ähnlichkeitsmaß Vektoren erstellt, die eigentliche Ähnlichkeitsbestimmung zwischen den Dokumenten erfolgt aber mit der Cosinusähnlichkeit.
2 Vgl. Cilibrasi u.a. (2005), S. 1528-1530.

samkeiten beide Texte haben, desto stärker kann ihre Konkatenation komprimiert werden. Die Autoren verwenden in den Experimenten nicht nur textuelle Daten, sondern auch andere Daten.[1] Da auch Suffix Arrays in der Kompression aufgrund ihrer Eigenschaft der lexikografischen Sortierung sehr gut eingesetzt werden können[2], bieten sich hier weitere Forschungsmöglichkeiten.

- Normalized Google Distance (NGD)[3]
  Dieses Ähnlichkeitsmaß erweitert die NCD, indem nicht mehr die Daten an sich komprimiert werden, sondern sie werden ersetzt durch ein so genanntes „... *Google event* ...“[4]. Das bedeutet, man sucht die zu vergleichenden Daten mit Google[5], ersetzt die zurückgelieferte Menge an Seiten durch eine so genannte Google „... *code-word length* ...“[6] und berechnet daraus eine Ähnlichkeit zwischen den zu vergleichenden Daten. Die Autoren behaupten, dass sich so auch semantische Ähnlichkeiten berücksichtigen lassen, da die gesuchten Terme in verschiedenen Kontexten gefunden werden.[7] In den durchgeführten Experimenten verwenden die Autoren das Ähnlichkeitsmaß beim Clustern und Klassifizieren.

Der letzte Ansatz lässt wenig Verbindung zu Suffix Arrays und TSF-Features erkennen. Jedoch könnten die Repräsentanten, nach denen gesucht wird, durch das vorgestellte Verfahren in Verbindung mit Featureselektion ermittelt werden. Anschließend könnte die NGD als Ähnlichkeitsmaßstab für die weitere Verarbeitung genutzt werden. So würde, laut Cilibrasi u.a. (2007), auch ein semantischer Vergleich einbezogen. Auch dieses Einsatzszenario für TSF-Features zeigt weiteren Forschungsbedarf auf.

1 Vgl. Cilibrasi u.a. (2005), S. 1539-1541.
2 Vgl. Adjeroh u.a. (2008), S. 51.
3 Vgl. Cilibrasi u.a. (2007), S. 374 f.
4 Cilibrasi u.a. (2007), S. 374.
5 An dieser Stelle weichen die Autoren nach Meinung der Verfasserin dieser Arbeit von einem feature-freien Ansatz ab, da bspw. nicht mit dem gesamten Text gesucht wird, sondern mit Suchbegriffen. Das sind Repräsentanten des gesamten Textes, also Features.
6 Cilibrasi u.a. (2007), S. 374.
7 Vgl. Cilibrasi u.a. (2007), S. 376.

Literaturverzeichnis

Abebe u.a. (2010)

Abebe, Surafel Lemma und Tonella, Paolo: *Natural Language Parsing of Program Element Names for Concept Extraction*. In: *IEEE 18th International Conference on Program Comprehension (ICPC)*. 2010, S. 156–159.

Abiteboul u.a. (2000)

Abiteboul, Serge; Buneman, Peter und Suciu, Dan: *Data on the Web: From Relations to Semistructured Data and XML*. San Francisco, CA: Kaufmann. 2000.

Abouelhoda u.a. (2004)

Abouelhoda, Mohamed Ibrahim; Kurtz, Stefan und Ohlebusch, Enno: *Replacing suffix trees with enhanced suffix arrays*. In: *Journal of Discrete Algorithms,* Jg. 2 (2004), S. 53–86.

Adjeroh u.a. (2008)

Adjeroh, Donald; Bell, Tim und Mukherjee, Amar: *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. New York, NY: Springer Science+Business Media LLC. 2008.

Aggarwal u.a. (2012)

Aggarwal, Charu C. und Zhai, ChengXiang: *A Survey of Text Classification Algorithms*. In: *Mining Text Data*. Hrsg. von Aggarwal, Charu C. und Zhai, ChengXiang. New York, NY: Springer. 2012, S. 165–222.

Aho u.a. (1974)

Aho, Alfred V.; Hopcroft, John E. und Ullman, Jeffrey D.: *The design and analysis of computer algorithms*. Addison-Wesley series in computer science and information processing. Reading, MA: Addison-Wesley. 1974.

Alfaro u.a. (2011)

Alfaro, Rodrigo und Allende, Héctor: *Text Representation in Multi-label Classification: Two New Input Representations*. In: *Adaptive and Natural Computing Algorithms*. Hrsg. von Dobnikar, Andrej; Lotrič, Uroš und Šter, Branko. Bd. 6594. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. 2011, S. 61–70.

Arthur u.a. (2005)

Arthur, David und Vassilvitskii, Sergei: *On the Worst-Case Complexity of the k-means Method*. Technischer Bericht 2005-34. Stanford InfoLab. 2005.

Baeza-Yates u.a. (1999)

Baeza-Yates, Ricardo und Ribeiro-Neto, Berthier: *Modern Information Retrieval*. Harlow: Pearson Addison-Wesley. 1999.

Balzert (1999)

Balzert, Helmut: *Lehrbuch Grundlagen der Informatik: Konzepte und Notationen in UML, Java und C++, Algorithmik und Software-Technik, Anwendungen*. Lehrbücher der Informatik. Heidelberg: Spektrum Akademischer Verlag. 1999.

Bayer (2010)

Bayer, Martin: *BI wird schlauer, mobiler und vorausschauender*. In: *Computerwoche,* Jg. 2010, Heft 39, S. 12–15. Im Internet unter der URL: http://www.computerwoche.de/software/bi-ecm/2354225/ (zuletzt besucht am: 03. 09. 2012).

Bayes u.a. (1763)

Bayes und Price: *An Essay towards Solving a Problem in the Doctrine of Chances. By the Late Rev. Mr. Bayes, F. R. S. Communicated by Mr. Price, in a Letter to John Canton, A. M. F. R. S.* In: *Philosophical Transactions,* Jg. 53 (1763), S. 370–418.

Bellet (2009)

Bellet, Aurélien: *Describing and testing two new linear Suffix Array Construction Algorithms*. Technischer Bericht. Department of Computing & Software, McMaster University, Hamilton, Ontario. 2009.

Bengio (2003)

Bengio, Yoshua; Ducharme, Réjean; Vincent, Pascal und Jauvin, Christian: *A Neural Probabilistic Language Model*. In: *Journal of Machine Learning Research,* Jg. 3 (2003), S. 1137–1155.

Bennett u.a. (2000)

Bennett, Kristin P. und Campbell, Colin: *Support Vector Machines: Hype or Hallelujah?* In: *SIGKDD Explorations Newsletter,* Jg. 2, Heft 2 (2000), S. 1–13.

Bird (2006)

Bird, Steven: *NLTK: The Natural Language Toolkit*. In: *ACL 2006, 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference, Sydney, Australia, 17-21 July 2006*. Sydney, Australia: The Association for Computer Linguistics. 2006, S. 69–72.

Bird u.a. (2009)

Bird, Steven; Klein, Ewan und Loper, Edward: *NLTK: Natural Language Toolkit Development*. Hrsg. von Bird, Steven; Klein, Ewan und Loper, Ed. Open-

Source-Software Version 0.9.9. Im Internet unter der URL: http://code.google.com/p/nltk/downloads/list (zuletzt besucht am: 16. 08. 2012).

Blum (2004)

Blum, Norbert: *Algorithmen und Datenstrukturen: Eine anwendungsorientierte Einführung*. München, Wien: Oldenbourg. 2004.

Blumberg u.a. (2003)

Blumberg, Robert und Atre, Shaku: *The Problem with Unstructured Data*. In: *DMReview,* Jg. 2003, S. 42–46. Im Internet unter der URL: http://soquelgroup.com/Articles/dmreview_0203_problem.pdf (zuletzt besucht am: 03. 09. 2012).

Bock (2008)

Bock, Hans-Hermann: *Origins and extensions of the k-means algorithm in cluster analysis*. In: *Electronic Journ@l for History of Probability and Statistics,* Jg. 4, Heft 2 (2008), S. 1–18.

Boisot u.a. (2004)

Boisot, Max und Canals, Agustí: *Data, information and knowledge: have we got it right?* In: *Journal of Evolutionary Economics,* Jg. 14 (2004), S. 43–67.

Bonin u.a. (2010)

Bonin, Francesca; Dell'Orletta, Felice; Montemagni, Simonetta und Venturi, Giulia: *A Contrastive Approach to Multi-word Term Extraction from Domain Corpora*. In: *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*. Hrsg. von Calzolari, Nicoletta (Conference Chair); Choukri, Khalid; Maegaard, Bente; Mariani, Joseph; Odijk, Jan; Piperidis, Stelios; Rosner, Mike und Tapias, Daniel. Valletta: European Language Resources Association (ELRA). 2010, S. 3222–3229.

Bortz u.a. (2002)

Bortz, Jürgen und Döring, Nicola: *Forschungsmethoden und Evaluation für Human- und Sozialwissenschaftler*. 3. Auflage. Springer-Lehrbuch. Heidelberg: Springer. 2002.

Bosma u.a. (2010)

Bosma, Wauter und Vossen, Piek: *Bootstrapping language-neutral term extraction*. In: *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*. Hrsg. von Calzolari, Nicoletta (Conference Chair); Choukri, Khalid; Maegaard, Bente; Mariani, Joseph; Odijk, Jan; Piperidis, Stelios; Rosner, Mike und Tapias, Daniel. Valletta: European Language Resources Association (ELRA). 2010, S. 2277–2282.

Böttcher (2006)

Böttcher, Axel: *Rechneraufbau und Rechnerarchitektur*. eXamen.press. Berlin, Heidelberg: Springer. 2006.

Breiman u.a. (1993)

Breiman, Leo; Friedman, Jerome H.; Olshen, Richard A. und Stone, Charles J.: *Classification and regression trees*. Boca Raton, FL: Chapman & Hall/CRC. 1993.

Bronstein u.a. (2005)

Bronstein, I. N. und Semendjajew, K. A.: *Taschenbuch der Mathematik*. 6. Aufl. Frankfurt am Main: Deutsch. 2005.

Brown u.a. (1990)

Brown, Peter F.; Cocke, John; Della Pietra, Stephen A.; Della Pietra, Vincent J.; Jelinek, Fredrick; Lafferty, John D.; Mercer, Robert L. und Roossin, Paul S.: *A statistical approach to machine translation*. In: *Computational Linguistics,* Jg. 16, Heft 2 (1990), S. 79–85.

Bruce u.a. (2004)

Bruce, Harry; Jones, William und Dumais, Susan: *Information behaviour that keeps found things found*. In: *Information Research,* Jg. 10, Heft 1 (2004). Paper 207. Im Internet unter der URL: http://informationr.net/ir/10-1/paper207.html (zuletzt besucht am: 21. 03. 2011).

Burrows u.a. (1994)

Burrows, M. und Wheeler, D. J.: *A Block-sorting Lossless Data Compression Algorithm*. Technischer Bericht. Palo Alto, CA: Digital Equipment Corporation. 1994.

Cha (2007)

Cha, Sung-Hyuk: *Comprehensive Survey on Distance / Similarity Measures between Probability Density Functions*. In: *International Journal of Mathematical Models and Methods in Applied Sciences,* Jg. 1, Heft 4 (2007), S. 300–307.

Chim u.a. (2007)

Chim, Hung und Deng, Xiaotie: *A New Suffix Tree Similarity Measure for Document Clustering*. In: *Proceedings of the 16th international conference on World Wide Web*. WWW '07. New York, NY: ACM. 2007, S. 121–130.

Choi u.a. (2010)

Choi, Seung-Seok; Cha, Sung-Hyuk und Tappert, Charles C.: *A Survey of Binary Similarity and Distance Measures*. In: *Journal of Systemics, Cybernetics and Informatics,* Jg. 8, Heft 1 (2010), S. 43–48.

Cilibrasi u.a. (2007)

Cilibrasi, Rudi L. und Vitányi, Paul M.B: *The Google Similarity Distance*. In: *IEEE Transactions on Knowledge and Data Engineering,* Jg. 19, Heft 3 (2007), S. 370–383.

Cilibrasi u.a. (2005)

Cilibrasi, Rudi und Vitányi, Paul M.B: *Clustering by Compression*. In: *IEEE Transactions on Information Theory,* Jg. 51, Heft 4 (2005), S. 1523–1545.

Cormen u.a. (2009)

Cormen, Thomas H.; Leiserson, Charles Eric; Rivest, Ronald Linn und Stein, Clifford: *Introduction to algorithms*. 3. Aufl. Cambridge, MA: MIT-Press. 2009.

Cortes u.a. (1995)

Cortes, Corinna und Vapnik, Vladimir: *Support-Vector Networks*. In: *Machine Learning,* Jg. 20 (1995), S. 273–297.

Cristianini u.a. (2002)

Cristianini, Nello und Shawe-Taylor, John: *An Introduction to Support Vector Machines: and other kernel-based learning methods*. Cambridge: Cambridge University Press. 2002.

Croft u.a. (1991)

Croft, W. Bruce; Turtle, Howard R. und Lewis, David D.: *The Use of Phrases and Structured Queries in Information Retrieval*. In: *SIGIR '91: Proceedings of the 14th annual international ACM SIGIR conference on Research and development in information retrieval*. New York, NY: ACM. 1991, S. 32–45.

Davenport u.a. (1998)

Davenport, Thomas H. und Prusak, Laurence: *Working knowledge: How organizations manage what they know*. Boston, MA: Harvard Business School Press. 1998.

Davenport u.a. (2000)

Davenport, Thomas H. und Prusak, Lawrence: *Working Knowledge: How Organizations Manage What They Know*. In: *Ubiquity,* Jg. 2000, S. 1–15. Im Internet unter der URL: http://wang.ist.psu.edu/course/05/IST597/papers/Davenport_know.pdf (zuletzt besucht am: 10. 04. 2012).

Deiser (2010)

Deiser, Oliver: *Einführung in die Mengenlehre: Die Mengenlehre Georg Cantors und ihre Axiomatisierung durch Ernst Zermelo*. 3. Aufl. Berlin, Heidelberg: Springer. 2010.

Dembczynski u.a. (2010)

Dembczynski, Krzysztof; Cheng, Weiwei und Hüllermeier, Eyke: *Bayes Op-*

timal Multilabel Classification via Probabilistic Classifier Chains. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*. Hrsg. von Fürnkranz, Johannes und Joachims, Thorsten. Omnipress. 2010, S. 279–286.

Deutsches Institut für Normung e.V.(1987)

Deutsches Institut für Normung e.V.: *Klassifikationssysteme*. Norm DIN 32705. Berlin: Beuth. 1987.

Dhillon u.a. (2004)

Dhillon, Inderjit; Kogan, Jacob und Nicholas, Charles: *Feature Selection and Document Clustering*. In: *Survey of Text Mining*. Hrsg. von Berry, Michael W. New York: Springer. 2004, S. 73–100.

Dinh u.a. (2011)

Dinh, Duy und Tamine, Lynda: *Biomedical concept extraction based on combining the content-based and word order similarities*. In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. SAC '11. New York, NY: ACM. 2011, S. 1159–1163.

Dom (2001)

Dom, Byron E.: *An Information-Theoretic External Cluster-Validity Measure*. Research Report RJ 10219. Yorktown Heights, NY, San Jose, CA, Zurich: International Business Machines Corporation (IBM). 2001.

Dörre u.a. (1999)

Dörre, Jochen; Gerstl, Peter und Seiffert, Roland: *Text Mining: Finding Nuggets in Mountains of Textual Data*. In: *KDD '99: Proceedings of the fifth Association for Computing Machinery (ACM) Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD) International Conference on Knowledge Discovery and Data Mining*. ACM. 1999, S. 398–401.

Echarte u.a. (2011)

Echarte, F.; Astrain, J. J.; Córdoba, A.; Villadangos, J. und Labat, A.: *A Method for the Classification of Folksonomy Resources*. In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. SAC '11. New York, NY: ACM. 2011, S. 1675–1680.

Eckey u.a. (2002)

Eckey, Hans-Friedrich; Kosfeld, Reinhold und Rengers, Martina: *Multivariate Statistik: Grundlagen, Methoden, Beispiele*. 1. Aufl. Wiesbaden: Gabler. 2002.

Esuli u.a. (2008)

Esuli, Andrea; Fagni, Tiziano und Sebastiani, Fabrizio: *Boosting multi-label*

hierarchical text categorization. In: *Information Retrieval,* Jg. 11 (2008), S. 287–313.

Feldman u.a. (2002)

Feldman, Ronen; Aumann, Yonatan; Finkelstein-Landau, Michal; Hurvitz, Eyal; Regev, Yizhar und Yaroshevich, Ariel: *A Comparative Study of Information Extraction Strategies*. In: *Computational Linguistics and Intelligent Text Processing*. Hrsg. von Gelbukh, Alexander. Bd. 2276. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. 2002, S. 349–359.

Feldman u.a. (1998)

Feldman, Ronen; Fresko, Moshe; Hirsh, Haym; Aumann, Yonatan; Liphstat, Orly; Schler, Yonatan und Rajman, Martin: *Knowledge Management: A Text Mining Approach*. In: *PAKM 98 Practical Aspects of Knowledge Management*. Bd. 13. CEUR Workshop Proceedings. 1998, pages.

Feldman u.a. (2007)

Feldman, Ronen und Sanger, James: *The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. Cambridge, New York: Cambridge University Press. 2007.

Feldman u.a. (2000)

Feldman, Susan und Sherman, Chris: *The High Cost of Not Finding Information*. Hrsg. von International Data Corporation (IDC). White Paper. 2000. Im Internet unter der URL: http://ejitime.com/materials/IDC%20on%20The%20High%20Cost%20Of%20Not%20Finding%20Information.pdf (zuletzt besucht am: 04. 09. 2012).

Ferragina u.a. (2000)

Ferragina, Paolo und Manzini, Giovanni: *Opportunistic Data Structures with Applications*. In: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. Washington, DC: IEEE Computer Society. 2000, S. 390–398.

Ferragina u.a. (2001)

Ferragina, Paolo und Manzini, Giovanni: *An experimental study of an opportunistic index*. In: *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*. SODA '01. Philadelphia, PA: Society for Industrial und Applied Mathematics. 2001, S. 269–278.

Ferragina u.a. (2005)

Ferragina, Paolo und Manzini, Giovanni: *Indexing Compressed Text*. In: *Journal of the Association for Computing Machinery,* Jg. 52, Heft 4 (2005), S. 552–581.

Fischer u.a. (2005)

Fischer, Johannes; Heun, Volker und Kramer, Stefan: *Fast Frequent String Mining Using Suffix Arrays*. In: *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005), 27-30 November 2005, Houston, Texas, USA*. IEEE Computer Society. 2005, S. 609–612.

Forster (2006)

Forster, Richard: *Document Clustering In Large German Corpora Using Natural Language Processing*. Dissertation. University of Zürich. 2006.

Frigui u.a. (2004)

Frigui, Hichem und Nasraoui, Olfa: *Simultaneous Clustering and Dynamic Keyword Weighting for Text Documents*. In: *Survey of Text Mining*. Hrsg. von Berry, Michael W. New York, NY: Springer. 2004, S. 45–72.

Furguson (2005)

Furguson, Charles H.: *Ein bisschen böse*. In: *Technology Review,* Jg. 2005, Heft 2, S. 38–47.

Ghamrawi u.a. (2005)

Ghamrawi, Nadia und McCallum, Andrew: *Collective Multi-Label Classification*. In: *CIKM '05: Proceedings of the 14th ACM International Conference on Information and Knowledge Management*. New York, NY: ACM. 2005, S. 195–200.

Giegerich u.a. (1999)

Giegerich, Robert; Kurtz, Stefan und Stoye, Jens: *Efficient Implementation of Lazy Suffix Trees*. In: *Proceedings of the 3rd International Workshop on Algorithm Engineering*. Workshop on Algorithm Engineering (WAE) '99. London: Springer. 1999, S. 30–42.

Gonnet (1983)

Gonnet, Gaston H.: *Unstructured Data Bases or Very Efficient Text Searching: Extended Abstract*. In: *Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*. PODS '83. New York, NY: ACM. 1983, S. 117–124.

Gonnet u.a. (1991)

Gonnet, Gastón H. und Baeza-Yates, Ricardo: *Handbook of Algorithms and Data Structures: In Pascal and C*. 2. Auflage, reprinted. International computer science series. Wokingham: Addison-Wesley. 1991.

Gonnet u.a. (1992)

Gonnet, Gaston H.; Baeza-Yates, Ricardo und Snider, Tim: *New Indices For Text: PAT Trees And PAT Arrays*. In: *Information Retrieval: Data Structures*

and Algorithms. Hrsg. von Frakes, William Bruce und Baeza-Yates, Ricardo. Englewood Cliffs, NJ: Prentice Hall. 1992.

Gopal u.a. (2011)

Gopal, Ram D.; Marsden, James R. und Vanthienen, Jan: *Information mining – Reflections on recent advancements and the road ahead in data, text, and media mining*. In: *Decision Support Systems,* Jg. 51, Heft 4 (2011), S. 727–731.

Grimes (2008)

Grimes, Seth: *Unstructured Data and the 80 Percent Rule*. Hrsg. von Clarabridge. Internetdokument. 2008. Im Internet unter der URL: http://clarabridge.com/default.aspx?tabid=137&ModuleID=635&ArticleID=551 (zuletzt besucht am: 03. 09. 2012).

Grossi u.a. (2000)

Grossi, Roberto und Vitter, Jeffrey Scott: *Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching (extended abstract)*. In: *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. STOC '00. New York, NY: ACM. 2000, S. 397–406.

Grossman u.a. (2004)

Grossman, David A. und Frieder, Ophir: *Information retrieval: Algorithms and Heuristics*. 2. Aufl. Bd. 15. Kluwer international series on information retrieval. Dordrecht: Springer. 2004.

Gusfield (1999)

Gusfield, Dan: *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. 1. Auflage. Cambridge u.a.: Cambridge University Press. 1999.

Handl (2010)

Handl, Andreas: *Ähnlichkeits- und Distanzmaße*. In: *Multivariate Analysemethoden*. Hrsg. von Dette, Holger und Härdle, Wolfgang. Statistik und ihre Anwendungen. Berlin, Heidelberg: Springer. 2010, S. 83–111.

Hawking u.a. (1995)

Hawking, David und Thistlewaite, Paul: *Proximity Operators - So Near And Yet So Far*. In: *4th Text Retrieval Conference (TREC-4)*. Hrsg. von Harmen, D. K. Special Publication 500-234, online publication. 1995, S. 131–143.

Hawkins (2004)

Hawkins, Douglas M.: *The Problem of Overfitting*. In: *Journal of Chemical Information and Computer Sciences,* Jg. 44, Heft 1 (2004), S. 1–12.

Hubwieser u.a. (2004)

Hubwieser, Peter und Aiglstorfer, Gerd: *Fundamente der Informatik: Ab-*

laufmodellierung, Algorithmen und Datenstrukturen. München, Wien: Oldenbourg. 2004.

Irani u.a. (2010)

Irani, Danesh; Webb, Steve; Pu, Calton und Li, Kang: *Study on Trend-Stuffing on Twitter through Text Classification*. 2010. Im Internet unter der URL: http://ceas.cc/2010/papers/Paper%2013.pdf (zuletzt besucht am: 12. 01. 2012).

Jain u.a. (1988)

Jain, Anil K. und Dubes, Richard C.: *Algorithms for Clustering Data*. Prentice-Hall Advanced Reference Series. Englewood Cliffs, NJ: Prentice-Hall. 1988.

Jain u.a. (1999)

Jain, Anil K.; Murty, M. Narasimha und Flynn, Patrick J.: *Data Clustering: A Review*. In: *Association for Computing Machinery (ACM) Computing Surveys,* Jg. 31, Heft 3 (1999), S. 264–323.

Jardine u.a. (1971)

Jardine, Nicholas und Rijsbergen, Cornelis J. van: *The use of hierarchic clustering in information retrieval*. In: *Information Storage and Retrieval,* Jg. 7, Heft 5 (1971), S. 217–240.

Jeon u.a. (2005)

Jeon, Jeong Eun; Park, Heejin und Kim, Dong Kyue: *Efficient Construction of Generalized Suffix Arrays by Merging Suffix Arrays*. In: *Journal of Korea Information Science Society (KISS): computer systems and technology,* Jg. 32, Heft 6 (2005), S. 268–278.

Ji u.a. (2008)

Ji, Shuiwang; Tang, Lei; Yu, Shipeng und Ye, Jieping: *Extracting Shared Subspace for Multi-label Classification*. In: *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. KDD '08. New York, NY: ACM. 2008, S. 381–389.

Joachims (1998)

Joachims, Thorsten: *Text categorization with Support Vector Machines: Learning with Many Relevant Features*. In: *Machine Learning: ECML-98*. Hrsg. von Nédellec, Claire und Rouveirol, Céline. Bd. 1398. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. 1998, S. 137–142.

Joachims (1999)

Joachims, Thorsten: *Transductive Inference for Text Classification using Support Vector Machines*. In: *Proceedings of the Sixteenth International Conference on Machine Learning*. ICML '99. San Francisco,CA: Morgan Kaufmann Publishers Inc. 1999, S. 200–209.

Joachims (2008a)

Joachims, Thorsten: *SVMlight*. 2008. Im Internet unter der URL: http://svmlight.joachims.org/ (zuletzt besucht am: 11. 01. 2012).

Joachims (2008b)

Joachims, Thorsten: *SVMmulticlass*. 2008. Im Internet unter der URL: http://svmlight.joachims.org/svm_multiclass.html (zuletzt besucht am: 11. 01. 2012).

Johnson u.a. (2006)

Johnson, David; Malhotra, Vishv und Vamplew, Peter: *More Effective Web Search Using Bigrams and Trigrams*. In: *Webology,* Jg. 3, Heft 4 (2006). Article 35. Im Internet unter der URL: http://www.webology.org/2006/v3n4/a35.html (zuletzt besucht am: 21. 08. 2012).

Kärkkäinen u.a. (2003)

Kärkkäinen, Juha und Sanders, Peter: *Simple Linear Work Suffix Array Construction*. In: *Proceedings of the 30th international conference on Automata, languages and programming*. ICALP'03. Berlin, Heidelberg: Springer. 2003, S. 943–955.

Kärkkäinen u.a. (2006)

Kärkkäinen, Juha; Sanders, Peter und Burkhardt, Stefan: *Linear Work Suffix Array Construction*. In: *Journal of the Association for Computing Machinery,* Jg. 53, Heft 6 (2006), S. 918–936.

Kehagias u.a. (2003)

Kehagias, Athanasios; Petridis, Vassilios; Kaburlasos, Vassilis G. und Fragkou, Pavlina: *A Comparison of Word- and Sense-Based Text Categorization Using Several Classification Algorithms*. In: *Journal of Intelligent Information Systems,* Jg. 21, Heft 3 (2003), S. 227–247.

Kim u.a. (2003)

Kim, Dong Kyue; Sim, Jeong Seop; Park, Heejin und Park, Kunsoo: *Linear-Time Construction of Suffix Arrays*. In: *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Morelia, Michocán, Mexico, June 25-27, 2003, Proceedings*. CPM'03. Berlin, Heidelberg: Springer. 2003, S. 186–199.

Klein u.a. (2008)

Klein, Robert und Steinhardt, Claudius: *Revenue Management: Grundlagen und Mathematische Methoden*. Berlin: Springer. 2008.

Ko u.a. (2003)

Ko, Pang und Aluru, Srinivas: *Space Efficient Linear Time Construction of Suffix Arrays*. In: *Combinatorial Pattern Matching, 14th Annual Symposium, CPM 2003, Morelia, Michocán, Mexico, June 25-27, 2003, Proceedings*. Hrsg. von Baeza-

Yates, Ricardo A.; Chávez, Edgar und Crochemore, Maxime. Bd. 2676. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. 2003, S. 200–210.

Ko u.a. (2005)

Ko, Pang und Aluru, Srinivas: *Space efficient linear time construction of suffix arrays.* In: *Journal of Discrete Algorithms,* Jg. 3 (2005), S. 143–156.

Koch (2010)

Koch, Daniel: *XML für Webentwickler: Ein praktischer Einstieg.* München: Hanser. 2010.

Koller u.a. (1997)

Koller, Daphne und Sahami, Mehran: *Hierarchically classifying documents using very few words.* In: *Proceedings of the Fourteenth International Conference on Machine Learning.* ICML '97. San Francisco, CA: Morgan Kaufmann Publishers Inc. 1997, S. 170–178.

Krcmar (2005)

Krcmar, Helmut: *Informationsmanagement.* 4. Aufl. Berlin, Heidelberg: Springer. 2005.

Kumaran u.a. (2012)

Kumaran, V. Senthil und Sankar, A.: *Expert locator using concept linking.* In: *International Journal of Computational Systems Engineering,* Jg. 1, Heft 1 (2012), S. 42–49.

Kurtz (1999)

Kurtz, Stefan: *Reducing the Space Requirement of Suffix Trees.* In: *Software Practice And Experience,* Jg. 29 (1999), S. 1149–1171.

Lance u.a. (1967)

Lance, G. N. und Williams, W. T.: *A general theory of classificatory sorting strategies: 1. Hierarchical systems.* In: *The Computer Journal,* Jg. 9, Heft 4 (1967), S. 373–380.

Larsen u.a. (1999)

Larsen, Bjornar und Aone, Chinatsu: *Fast and Effective Text Mining Using Linear-time Document Clustering.* In: *KDD '99: Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining.* New York, NY: ACM. 1999, S. 16–22.

Larsson u.a. (1999)

Larsson, N. Jesper und Sadakane, Kunihiko: *Faster Suffix Sorting.* Techreport. Department of Computer Science, Lund University. 1999. Im Internet unter der URL: http://www.larsson.dogma.net/ssrev-tr.pdf (zuletzt besucht am: 04. 09. 2012).

Lewis (1991)

Lewis, David D.: *Evaluating Text Categorization.* In: *Human Language Technologies (HLT) 91: Proceedings of the workshop on Speech and Natural Language.* Morristown, NJ: Association for Computational Linguistics. 1991, S. 312–318.

Lewis (2004)

Lewis, David D.: *RCV1-v2/LYRL2004: The LYRL2004 Distribution of the RCV1-v2 Text Categorization Test Collection.* Hrsg. von Lewis, David D. 2004. Im Internet unter der URL: http://www.ai.mit.edu/projects/jmlr/papers/volume5/lewis04a/lyrl2004_rcv1v2_README.htm (zuletzt besucht am: 15. 08. 2012).

Lewis u.a. (2004)

Lewis, David D.; Yang, Yiming; Rose, Tony G. und Li, Fan: *RCV1: A New Benchmark Collection for Text Categorization Research.* In: *Journal of Machine Learning Research,* Jg. 5 (2004), S. 361–397.

Li u.a. (2010)

Li, Chun sheng; Wang, Yaonan und Yang, Haidong: *Combining Fuzzy Partitions Using Fuzzy Majority Vote and KNN.* In: *Journal of Computers,* Jg. 5, Heft 5 (2010), S. 791–798.

Liu u.a. (2002)

Liu, Xin; Gong, Yihong; Xu, Wei und Zhu, Shenghuo: *Document Clustering with Cluster Refinement and Model Selection Capabilities.* In: *SIGIR '02: Proceedings of the 25th annual international Association for Computing Machinery (ACM) Special Interest Group on Information Retrieval (SIGIR) conference on Research and development in information retrieval.* New York, NY: Association for Computing Machinery (ACM). 2002, S. 191–198.

Lloyd (1982)

Lloyd, Stuart P.: *Least Squares Quantization in PCM.* In: *Institute of Electrical and Electronics Engineers (IEEE) Transactions on Information Theory,* Jg. 28, Heft 2 (1982), S. 129–137.

Logan (2012)

Logan, Robert K.: *What Is Information?: Why Is It Relativistic and What Is Its Relationship to Materiality, Meaning and Organization.* In: *Information,* Jg. 3, Heft 1 (2012), S. 68–91.

MacQueen (1967)

MacQueen, J. B.: *Some Methods for Classification and Analysis of Multivariate Observations.* In: *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability.* Hrsg. von Le Cam, Lucien M. und Neyman, Jerzy. Bd. 1. Statistics. Berkeley, CA: University of California Press. 1967, S. 281–297.

Majster u.a. (1979)

Majster, M. und Reiser, A.: *An efficient on-line position tree construction algorithm*. In: *Theoretical Computer Science 4th GI Conference*. Hrsg. von Weihrauch, K. Bd. 67. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. 1979, S. 190–198.

Malz (2001)

Malz, Helmut: *Rechnerarchitektur: Eine Einführung für Ingenieure und Informatiker*. uni-script. Braunschweig: Vieweg. 2001.

Manber u.a. (1990)

Manber, Udi und Myers, Gene: *Suffix Arrays: A New Method for On-Line String Searches*. In: *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '90. Philadelphia, PA: Society for Industrial und Applied Mathematics. 1990, S. 319–327.

Maniscalco u.a. (2006)

Maniscalco, Michael A. und Puglisi, Simon J.: *Faster Lightweight Suffix Array Construction*. In: *Proceedings of 17th Australasian Workshop on Combinatorial Algorithms*. Hrsg. von J. Ryan & Dafik. Ballavat, Victoria. 2006, S. 16–29.

Manning u.a. (2008)

Manning, Christopher D.; Raghavan, Prabhakar und Schütze, Hinrich: *Introduction to Information Retrieval*. Cambridge: Cambridge University Press. 2008.

Märtin (1994)

Märtin, Christian: *Rechnerarchitektur: Struktur, Organisation, Implementierungstechnik*. Hanser-Studienbücher der Informatik. München: Hanser. 1994.

McCreight (1976)

McCreight, Edward M.: *A Space-Economical Suffix Tree Construction Algorithm*. In: *Journal of the Association for Computing Machinery,* Jg. 23, Heft 2 (1976), S. 262–272.

Meier (2010)

Meier, Andreas: *Relationale und postrelationale Datenbanken*. 6. Aufl. Berlin, Heidelberg: Springer. 2010.

Mitchell (1997)

Mitchell, Tom Michael: *Machine learning*. International Edition. McGraw-Hill series in computer science. McGraw-Hill. 1997.

Moore (2011)

Moore, Andrew: *Decision Trees*. Internetdokument. ohne Jahr. Im Internet unter der URL: http://www.autonlab.org/tutorials/dtree18.pdf (zuletzt besucht am: 16. 08. 2012).

Mori (2010)

Mori, Yuta: *sais*. Open-Source-Software Java-Version. Im Internet unter der URL: http://sites.google.com/site/yuta256/sais (zuletzt besucht am: 10. 08. 2012).

Morrison (1968)

Morrison, Donald R.: *PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric*. In: *Journal of the Association for Computing Machinery,* Jg. 15, Heft 4 (1968), S. 514–534.

Narasimhamurthy (2005)

Narasimhamurthy, Anand: *Theoretical Bounds of Majority Voting Performance for a Binary Classification Problem*. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence,* Jg. 27, Heft 12 (2005), S. 1988–1995.

Navarro u.a. (2000)

Navarro, Gonzalo und Baeza-Yates, Ricardo: *A Hybrid Indexing Method for Approximate String Matching*. In: *Journal of Discrete Algorithms,* Jg. 1, Heft 1 (2000), S. 205–239.

NIST (2004)

NIST: *Reuters Corpora*. Hrsg. von National Institute of Standards and Technology (NIST). 2004. Im Internet unter der URL: http://trec.nist.gov/data/reuters/reuters.html (zuletzt besucht am: 15. 08. 2012).

Noble (2006)

Noble, William S.: *What is a support vector machine?* In: *Nature Biotechnology,* Jg. 24, Heft 12 (2006), S. 1565–1567.

Nong u.a. (2009)

Nong, Ge; Zhang, Sen und Chan, Wai Hong: *Linear Suffix Array Construction by Almost Pure Induced-Sorting*. In: *Proceedings of the 2009 Data Compression Conference*. Washington, DC: IEEE Computer Society. 2009, S. 193–202.

Norvig (2009)

Norvig, Peter: *Natural Language Corpus Data*. In: *Beautiful Data: The Stories Behind Elegant Data Solutions*. Hrsg. von Segaran, Toby und Hammerbacher, Jeff. Sebastopol, California: O'Reilly. 2009, S. 219–242.

Oberschelp u.a. (2000)

Oberschelp, Walter und Vossen, Gottfried: *Rechneraufbau und Rechnerstrukturen*. 8. Aufl. München: Oldenbourg. 2000.

Ottmann u.a. (2012)

Ottmann, Thomas und Widmayer, Peter: *Algorithmen und Datenstrukturen*. 5. Aufl. Heidelberg: Spektrum Akademischer Verlag. 2012.

Papka u.a. (1998)

Papka, Ron und Allan, James: *Document Classification using Multiword Features*. In: *Proceedings of the 1998 ACM CIKM International Conference on Information and Knowledge Management*. CIKM '98. New York, NY, USA: ACM. 1998, S. 124–131.

Paul u.a. (2006)

Paul, Topon Kumar; Hasegawa, Yoshihiko und Iba, Hitoshi: *Classification of Gene Expression Data by Majority Voting Genetic Programming Classifier*. In: *IEEE Congress on Evolutionary Computation, 2006. CEC 2006.* Institute of Electrical und Electronics Engineers. 2006, S. 2521–2528.

Pelleg u.a. (2000)

Pelleg, Dan und Moore, Andrew: *X-means: Extending K-means with Efficient Estimation of the Number of Clusters*. In: *Proceedings of the Seventeenth International Conference on Machine Learning*. San Francisco, CA: Morgan Kaufmann. 2000, S. 727–734.

Piao u.a. (2010)

Piao, Scott; Forth, Jamie; Gacitua, Ricardo; Whittle, Jon und Wiggins, Geraint: *Evaluating Tools for Automatic Concept Extraction: a Case Study from the Musicology Domain*. In: *Proceedings of The Digital Economy All Hands Meeting - Digital Futures 2010 Conference*. Internetdokument. Nottingham. 2010. Im Internet unter der URL: http://eprints.lancs.ac.uk/51585/1/p16_Piao.pdf.

Pomberger u.a. (2008)

Pomberger, Gustav und Dobler, Heinz: *Algorithmen und Datenstrukturen: Eine systematische Einführung in die Programmierung*. it-informatik. München: Pearson Studium. 2008.

Poulos u.a. (2005)

Poulos, Marios; Papavlasopoulos, Sozon und Chrissikopoulos, Vasilios: *A Text Categorization Technique based on a Numerical Conversion of a Symbolic Expression and an Onion Layers Algorithm*. In: *Journal of Digital Information,* Jg. 6, Heft 1 (2005).

Puglisi u.a. (2007)

Puglisi, Simon J.; Smyth, W. F. und Turpin, Andrew H.: *A Taxonomy of Suffix Array Construction Algorithms*. In: *Association for Computing Machinery (ACM) Computing Surveys,* Jg. 39, Heft 2 (2007), S. 1–31.

Python Software Foundation (2010)

Python Software Foundation: *Python*. Open-Source-Software. 2010. Im Internet unter der URL: http://www.python.org/ (zuletzt besucht am: 07. 08. 2012).

Quinlan (1986)

Quinlan, J. Ross: *Induction of Decision Trees*. In: *Machine Learning,* Jg. 1 (1986), S. 81–106.

Quinlan (1993)

Quinlan, J. Ross: *C4.5: Programs for machine learning*. San Mateo: Morgan Kaufmann. 1993.

Rand (1971)

Rand, William M.: *Objective Criteria for the Evaluation of Clustering Methods*. In: *Journal of the American Statistical Association,* Jg. 66, Heft 336 (1971), S. 846–850.

Rao (2003)

Rao, Ramana: *From Unstructured Data to Actionable Intelligence*. In: *IT Professional,* Jg. 5, Heft 6 (2003), S. 29–35.

Read u.a. (2008)

Read, Jesse; Pfahringer, Bernhard und Holmes, Geoff: *Multi-label Classification using Ensembles of Pruned Sets*. In: *Eighth IEEE International Conference on Data Mining, 2008. ICDM '08.* IEEE Computer Society. 2008, S. 995–1000.

Read u.a. (2009)

Read, Jesse; Pfahringer, Bernhard; Holmes, Geoff und Frank, Eibe: *Classifier Chains for Multi-label Classification*. In: *Machine Learning and Knowledge Discovery in Databases*. Hrsg. von Buntine, Wray; Grobelnik, Marko; Mladenic, Dunja und Shawe-Taylor, John. Bd. 5782. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. 2009, S. 254–269.

Rivera u.a. (2011)

Rivera, A. J.; Charte, F.; Pérez-Godoy, M. D. und del Jesus, María Jose: *Multi-label Testing for CO²RBFN: A first approach to the problem transformation methodology for multi-label classification*. In: *Advances in Computational Intelligence*. Hrsg. von Cabestany, Joan; Rojas, Ignacio und Joya, Gonzalo. Bd. 6691. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. 2011, S. 41–48.

Robb (2004)

Robb, Drew: *Text mining tools take on unstructured data*. Internetdokument. 2004. Im Internet unter der URL: http://www.computerworld.com/s/article/print/93968/Taming_Text (zuletzt besucht am: 04. 09. 2012).

Roja u.a. (2011)

Roja, M. Mani und Sawarkar, Sudhir: *A Hybrid Approach using Majority Voting for Signature Recognition*. In: *3rd International Conference on Electronics*

Computer Technology (ICECT), 2011. Bd. 2. Hong Kong: Institute of Electrical und Electronics Engineers. 2011, S. 354–358.

Russell u.a. (2003)

Russell, Stuart Jonathan und Norvig, Peter: *Artificial Intelligence: A Modern Approach*. 2nd edition, international edition. Prentice Hall series in artificial intelligence. Upper Saddle River, NJ: Prentice Hall. 2003.

Russom (2007)

Russom, Philip: *BI Search And Text Analytics: New Additions to the BI Technology Stack*. TDWI Best Practices Report. The Data Warehousing Institute. 2007.

Sadakane (1998)

Sadakane, Kunihiko: *A Fast Algorithm for Making Suffix Arrays and for Burrows-Wheeler Transformation*. In: *Proceedings of the IEEE Data Compression Conference*. IEEE Computer Society Press. 1998, S. 129–138.

Sajnani u.a. (2011)

Sajnani, Hitesh; Javanmardi, Sara; McDonald, David W. und Lopes, Cristina V.: *Multi-Label Classification of Short Text: A Study on Wikipedia Barnstars*. In: *AAAI Workshops*. Hrsg. von AAAI. 2011, S. 56–61. Im Internet unter der URL: http://aaai.org/ocs/index.php/WS/AAAIW11/paper/view/3909 (zuletzt besucht am: 03. 08. 2012).

Scholze-Stubenrecht u.a. (2006)

Scholze-Stubenrecht, Werner und Wermke, Matthias: *Duden - die deutsche Rechtschreibung*. 24. Aufl. Bd. 1. Der Duden. Mannheim: Dudenverlag. 2006.

Schürmann u.a. (2007)

Schürmann, Klaus-Bernd und Stoye, Jens: *An incomplex algorithm for fast suffix array construction*. In: *Software - Practice and Experience,* Jg. 37 (2007), S. 309–329.

Sebastiani (1999)

Sebastiani, Fabrizio: *A Tutorial on Automated Text Categorisation*. In: *Proceedings of the 1st Argentinian Symposium on Artificial Intelligence (ASAI'99)*. Hrsg. von Amandi, Analia und Zunino, Alejandro. Buenos Aires. 1999.

Sebastiani (2002)

Sebastiani, Fabrizio: *Machine Learning in Automated Text Categorization*. In: *Association for Computing Machinery (ACM) Computing Surveys,* Jg. 34, Heft 1 (2002), S. 1–47.

Segaran (2007)

Segaran, Toby: *Programming Collective Intelligence: Building Smart Web 2.0 Applications*. 1. Auflage. Sebastopol, CA: O'Reilly. 2007.

Shannon u.a. (1963)

SHANNON, Claude E und WEAVER, Warren: *The mathematical theory of communication*. Illini books edition. Urbana: University of Illinois Press. 1963.

Shi (1996)

SHI, Fei: *Suffix Arrays for Multiple Strings: A Method for On-Line Multiple String Searches*. In: *Concurrency and Parallelism, Programming, Networking, and Security*. Hrsg. von JAFFAR, Joxan und YAP, Roland. Bd. 1179. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. 1996, S. 11–22.

Silla u.a. (2011)

SILLA, Carlos und FREITAS, Alex: *A survey of hierarchical classification across different application domains*. In: *Data Mining and Knowledge Discovery,* Jg. 22 (2011), S. 31–72.

Simpson u.a. (2010)

SIMPSON, Jared T. und DURBIN, Richard: *Efficient construction of an assembly string graph using the FM-index*. In: *Bioinformatics,* Jg. 26, Heft 12 (2010), S. i367–i373.

Sirén (2009)

SIRÉN, Jouni: *Compressed Suffix Arrays for Massive Data*. In: *Proceedings of the 16th International Symposium on String Processing and Information Retrieval*. SPIRE '09. Berlin, Heidelberg: Springer. 2009, S. 63–74.

Slonim u.a. (2005)

SLONIM, Noam; ATWAL, Gurinder Singh; TKAČIK, Gašper und BIALEK, William: *Information-based clustering*. In: *Proceedings of the National Academy of Sciences of the United States of America,* Jg. 102, Heft 51 (2005), S. 18297–18302.

Solka (2008)

SOLKA, Jeffrey L.: *Text Data Mining: Theory and Methods*. In: *Statistics Surveys,* Jg. 2 (2008), S. 94–112.

Spangler u.a. (2007)

SPANGLER, Scott und KREULEN, Jeffrey: *Mining the Talk: Unlocking the Business Value in Unstructured Information*. Stoughton, MA: IBM Press. 2007.

Staud (2010)

STAUD, Josef L.: *Unternehmensmodellierung: Objektorientierte Theorie und Praxis mit UML 2.0*. Berlin, Heidelberg: Springer. 2010.

Stergios (2010)

STERGIOS, Gosia: *Information Overload: New Symptoms and New Solutions*. Hrsg. von KNOL. 2010. Im Internet unter der URL: http://knol.google.com/k/gosia-

stergios/information-overload-new-symptoms-and/2xds05o8u2fgp/11 (zuletzt besucht am: 18. 01. 2011).

Strehl (2002)

Strehl, Alexander: *Relationship-based Clustering and Cluster Ensembles for High-dimensional Data Mining*. Dissertation. The University of Texas at Austin. 2002.

Strehl u.a. (2000)

Strehl, Alexander; Ghosh, Joydeep und Mooney, Raymond: *Impact of Similarity Measures on Web-page Clustering*. In: *Workshop on Artificial Intelligence for Web Search Association for the Advancement of Artificial Intelligence (AAAI 2000)*. AAAI Press. 2000, S. 58–64.

Strzalkowski (1996)

Strzalkowski, Tomek: *Natural language information retrieval: TIPSTER-2 final report*. In: *Proceedings of a workshop on held at Vienna, Virginia: May 6-8, 1996*. TIPSTER '96. Stroudsburg, PA: Association for Computational Linguistics. 1996, S. 143–148.

Tan (1999)

Tan, Ah-Hwee: *Text Mining: The state of the art and the challenges*. In: *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'99) Workshop on Knowledge Discovery from Advanced Databases*. Beijing. 1999, S. 65–70.

Tata u.a. (2004)

Tata, Sandeep; Hankins, Richard A. und Patel, Jignesh M.: *Practical Suffix Tree Construction*. In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*. Hrsg. von Nascimento, Mario A.; Özsu, M. Tamer; Kossmann, Donald; Miller, Renée J.; Blakeley, José A. und Schiefer, K. Bernhard. Morgan Kaufman. 2004, S. 36–47.

Tsoumakas u.a. (2007b)

Tsoumakas, Grigorios und Vlahavas, Ioannis: *Random k-Labelsets: An Ensemble Method for Multilabel Classification*. In: *Proceedings of the 18th European Conference on Machine Learning (ECML 2007)*. Hrsg. von Kok, J.N.; Koronacki, J.; de Mantaras, R.L.; Matwin, S.; Mladenic, D. und Skowron, A. Bd. 4701. LNAI. Berlin, Heidelberg: Springer. 2007, S. 406–417.

Tuomi (1999)

Tuomi, Ilkka: *Data Is More Than Knowledge: Implications of the Reversed Knowledge Hierarchy for Knowledge Management and Organizational Memory*. In: *Pro-*

ceedings of the 32nd Annual Hawaii International Conference on System Sciences, 1999. HICSS-32. Hrsg. von IEEE Computer Society. Bd. 1. 1999, S. 1–12.

Ukkonen (1995)

Ukkonen, Esko: *On-Line Construction of Suffix Trees.* In: *Algorithmica,* Jg. 14 (1995), S. 249–260.

Ullenboom (2011)

Ullenboom, Christian: *Java ist auch eine Insel: Das umfassende Handbuch.* 9. Aufl. Bonn: Galileo Press. 2011.

Vapnik (1995)

Vapnik, Vladimir N.: *The Nature of Statistical Learning Theory.* Zitiert nach Joachims (1998), S. 137. New York, NY: Springer. 1995.

Vens u.a. (2008)

Vens, Celine; Struyf, Jan; Schietgat, Leander; Džeroski, Sašo und Blockeel, Hendrik: *Decision trees for hierarchical multi-label classification.* In: *Machine Learning,* Jg. 73 (2008), S. 185–214.

Visual Paradigm (2012)

Visual Paradigm: *VP Gallery Activity Diagram.* Im Internet unter der URL: http://www.visual-paradigm.com/VPGallery/diagrams/Activity.html (zuletzt besucht am: 14. 08. 2012).

Voorhees (1986)

Voorhees, Ellen M.: *Implementing agglomerative hierarchic clustering algorithms for use in document retrieval.* In: *Information Processing & Management,* Jg. 22, Heft 6 (1986), S. 465–476.

Weiner (1973)

Weiner, Peter: *Linear pattern matching algorithms.* In: *IEEE Conference Record of 14th Annual Symposium on Switching and Automata Theory.* Northridge, CA: IEEE Computer Society. 1973, S. 1–11.

Weiss u.a. (2005)

Weiss, Sholom M.; Indurkhya, Nitin; Zhang, Tong und Damerau, Fred J.: *Text Mining: Predictive Methods for Analyzing Unstructured Information.* New York: Springer. 2005.

Willett (1988)

Willett, Peter: *Recent trends in hierarchic document clustering: a critical review.* In: *Information Processing and Management,* Jg. 24, Heft 5 (1988), S. 577–597.

Wilson u.a. (2011)

Wilson, Theresa und Hofer, Gregor: *Using Linguistic and Vocal Expressiveness*

in Social Role Recognition. In: *Proceedings of the 16th International Conference on Intelligent User Interfaces*. IUI '11. New York, NY: ACM. 2011, S. 419–422.

World-Wide Web Consortium (2004)

World-Wide Web Consortium: *Document Object Model Core*. Hrsg. von Le Hors, Arnaud (IBM); Le Hégaret, Philippe (W3C); Nicol, Gavin (Inso EPS (for DOM Level 1)); Wood, Lauren (SoftQuad Inc (for DOM Level 1)); Champion, Mike (Arbortext and Software AG (for DOM Level 1. from November 20 1997)) und Byrne, Steve (JavaSoft (for DOM Level 1. until November 19 1997)). 2004. Im Internet unter der URL: http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/core.html (zuletzt besucht am: 04. 08. 2012).

World-Wide Web Consortium (2008)

World-Wide Web Consortium: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Hrsg. von Bray, Tim (Textuality and Netscape); Paoli, Jean (Microsoft); Sperberg-McQueen, C. M. (W3C); Maler, Eve (Sun Microsystems Inc) und Yergeau, François. 2008. Im Internet unter der URL: http://www.w3.org/TR/2008/REC-xml-20081126/#dt-xml-proc (zuletzt besucht am: 04. 08. 2012).

Yamamoto u.a. (2001)

Yamamoto, Mikio und Church, Kenneth Ward: *Using Suffix Arrays to Compute Term Frequency and Document Frequency for All Substrings in a Corpus*. In: *Computational Linguistics,* Jg. 27, Heft 1 (2001), S. 1–30.

Yang (2006)

Yang, Hsin-Chang: *A method for automatic construction of learning contents in semantic web by a text mining approach*. In: *International Journal of Knowledge and Learning,* Jg. 2, Heft 1 (2006), S. 89–105.

Yang (1997)

Yang, Yiming: *An Evaluation of Statistical Approaches to Text Categorization*. In: *Journal of Information Retrieval,* Jg. 1 (1997), S. 69–90.

Younes (2011)

Younes, Zoulficar; Abdallah, Fahed; Denoeux, Thierry und Snoussi, Hichem: *A Dependent Multilabel Classification Method Derived from the k-Nearest Neighbor Rule*. In: *EURASIP Journal on Advances in Signal Processing,* Jg. 2011. Article ID 645964.

Zamir u.a. (1998)

Zamir, Oren und Etzioni, Oren: *Web Document Clustering: A Feasibility Demonstration*. In: *SIGIR '98: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*. New York, NY: ACM. 1998, S. 46–54.

Zeldes u.a. (2007)

Zeldes, Nathan; Sward, David und Louchheim, Sigal: *Infomania: Why we can't afford to ignore it any longer*. In: *First Monday[Online],* Jg. 12, Heft 8 (2007). Im Internet unter der URL: http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/1973/1848 (zuletzt besucht am: 04. 08. 2012).

Zhang u.a. (2005)

Zhang, Min-Ling und Zhou, Zhi-Hua: *A k-Nearest Neighbor Based Algorithm for Multi-label Classification*. In: *2005 IEEE International Conference on Granular Computing*. Bd. 2. 2005, S. 718–721.

Zhang u.a. (2007)

Zhang, Wen; Yoshida, Taketoshi und Tang, Xijin: *Text Classification using Multi-word Features*. In: *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Montréal, Canada, 7-10 October 2007*. IEEE. 2007, S. 3519–3524.

Zhao u.a. (2002)

Zhao, Ying und Karypis, George: *Criterion Functions for Document Clustering: Experiments and Analysis*. Technischer Bericht #01-40. Department of Computer Science, University of Minnesota, MN. 2002.

Zhu u.a. (2011)

Zhu, Lvxing; Chen, Youguang und Zhu, Min: *Layout Recognition of Multi-Page Document Based on Naive Bayes*. In: *2011 International Conference on Multimedia Technology (ICMT)*. Institute of Electrical und Electronics Engineers. 2011, S. 568–571.

Anhang A

Umformung der Formel zur Berechnung der Wahrscheinlichkeit für eine Klasse für ein Dokument

Es soll gelten:

$$P(L_j|d_i) = \frac{\prod_{r=1}^{|F|}\left[P(f_r=0|L_j) * \left(\frac{P(f_r=1|L_j)}{P(f_r=0|L_j)}\right)^{f_{r_{d_i}}}\right] * P(L_j)}{P(d_i)} =$$
$$\frac{1}{P(d_i)} \exp\left(\sum_{r=1}^{|F|} \ln\left(\frac{P(f_r=1|L_j)}{P(f_r=0|L_j)}\right) * f_{r_{d_i}} + \ln(P(L_j)) + \sum_{r=1}^{|F|} \ln(P(f_r=0|L_j))\right)$$

Unter Verwendung der Formel $a^x = \exp(x * \ln a)$[1] wird der erste Umformungsschritt durchgeführt. Es ergibt sich Folgendes:

$$P(L_j|d_i) = \frac{\prod_{r=1}^{|F|}[P(f_r=0|L_j)] * \prod_{r=1}^{|F|}\left[\left(\frac{P(f_r=1|L_j)}{P(f_r=0|L_j)}\right)^{f_{r_{d_i}}}\right] * P(L_j)}{P(d_i)}$$
$$= \frac{1}{P(d_i)} \exp\left(1 * \ln\left(\prod_{r=1}^{|F|}[P(f_r=0|L_j)]\right)\right) *$$
$$\exp\left(f_{r_{d_i}} * \ln\left(\prod_{r=1}^{|F|}\left[\left(\frac{P(f_r=1|L_j)}{P(f_r=0|L_j)}\right)\right]\right)\right) * \exp\left(1 * \ln\left(P(L_j)\right)\right)$$

Mit Hilfe der Formel $\ln(\prod_{i=1}^{n} x_i) = \sum_{i=1}^{n} \ln(x_i)$[2] erfolgt der nächste Umformungsschritt.

$$P(L_j|d_i) = \frac{1}{P(d_i)} \exp\left(\sum_{r=1}^{|F|} \ln\left(P(f_r=0|L_j)\right)\right) *$$
$$\exp\left(f_{r_{d_i}} * \sum_{r=1}^{|F|} \ln\left(\frac{P(f_r=1|L_j)}{P(f_r=0|L_j)}\right)\right) * \exp\left(\ln\left(P(L_j)\right)\right)$$

1 Vgl. Bronstein u.a. (2005), S. 721.

2 Die Umformung ist eine Abwandlung der Formel: $\ln(xy) = \ln(x) + \ln(y)$, vgl. Bronstein u.a. (2005), S. 9.

Zuletzt verwendet man die Formel $\exp(x) * \exp(y) = \exp(x+y)$[1].

$$P(L_j|d_i) = \frac{1}{P(d_i)} \exp\left(\sum_{r=1}^{|F|} \ln\left(P(f_r = 0|L_j)\right) + f_{r_{d_i}} * \sum_{r=1}^{|F|} \ln\left(\frac{P(f_r = 1|L_j)}{P(f_r = 0|L_j)}\right) + \ln\left(P(L_j)\right)\right)$$

Das entspricht, nach Umstellung, der oben angegebenen Formel.

1 Vgl. Bronstein u.a. (2005), S. 8.

Anhang B

Herleitung der verwendeten Formel für das F-Maß

Ausgehend vom harmonischen Mittel aus Genauigkeit und Trefferquote

$$\frac{2}{\frac{1}{Prec}+\frac{1}{Rec}}$$

werden nicht nur die beiden Werte betrachtet, sondern auch ihre Gewichtung. Für die Genauigkeit wird keine Gewichtung eingeführt, d.h., es bleibt dabei, dass sie einmal in das harmonische Mittel einfließt. Für die Trefferquote wird der Gewichtungsfaktor β eingeführt, der hier quadratisch einfließt. Das bedeutet, die Trefferquote fließt nicht einmal in das harmonische Mittel ein, sondern β^2-Mal. Mathematisch bedeutet das, dass der Gewichtungsfaktor β so in der ursprünglichen Formel ergänzt werden muss, dass die ursprüngliche Formel nicht verändert wird, wenn $\beta = 1$ ist. Daher wird die Formel wie folgt geändert:

$$\frac{\beta^2+1}{\frac{1}{Prec}+\beta^2*\frac{1}{Rec}}$$

Nun folgt eine Umformung, um nicht mit den Kehrwerten von Genauigkeit und Trefferquote rechnen zu müssen:

$$\begin{aligned} F_\beta &= \frac{\beta^2+1}{\frac{1}{Prec}+\beta^2*\frac{1}{Rec}} \\ &= \frac{\beta^2+1}{\frac{Rec}{Prec\ Rec}+\beta^2*\frac{Prec}{Prec\ Rec}} \\ &= \frac{\beta^2+1}{\frac{Rec+\beta^2*Prec}{Prec\ Rec}} \\ &= \frac{(\beta^2+1)Prec\ Rec}{\beta^2 Prec\ +\ Rec} \end{aligned}$$

Insgesamt ergibt sich daraus, dass bei $\beta = 1$ das *F-Maß* das harmonische Mittel aus Genauigkeit und Trefferquote ist und diese somit gleichgewichtet. Andere Werte für β führen zu einer stärkeren Gewichtung entweder der Genauigkeit oder der Trefferquote, siehe ab S. 76 der vorliegenden Arbeit.

Anhang C

Herleitung der verwendeten Formel für den Rand Index

C.1 Ursprüngliche Formel

Der Rand Index[1] beruht auf einem Vergleich zwischen zwei verschiedenen Clusterings, Y und Y', die auf den gleichen Daten basieren. Um sie vergleichen zu können, werden für jedes der zwei Ergebnisse dessen Cluster durchnummeriert und die Anzahl der Daten gezählt, die im i-ten Cluster von Y und im j-ten Cluster von Y vorhanden sind. Diese Anzahl wird mit n_{ij} bezeichnet.
Für das in Rand (1971)[2] zu findende Beispiel heißt das folgendes:

- $Y = \{(a, b, c), (d, e, f)\}$, also das Clustering Y, ordnet die sechs Daten *a, b, c, d, e, f* zwei Clustern zu, wobei sich a, b und c im ersten Cluster und d, e und f im zweiten Cluster befinden.

- $Y' = \{(a, b), (c, d, e), (f)\}$, also das Clustering Y', ordnet die sechs Daten a, b, c, d, e, f drei Clustern zu, wobei sich a und b im ersten Cluster, c, d und e im zweiten Cluster und f im dritten Cluster befinden.

- Y besteht also aus den Clustern Y_1 und Y_2, d.h., $i \in \{1, 2\}$, und Y' besteht aus den Clustern Y'_1, Y'_2 und Y'_3, d.h., $j \in \{1, 2, 3\}$.

Um die Anzahlen, also die n_{ij} zu bestimmen, vergleicht man die Daten auf ihr Vorhandensein in den beiden Clusterings. Das bedeutet beispielsweise, bei der Betrachtung von Y_2, dann ist $i = 2$, und von Y'_2, dann ist $j = 2$, stellt man fest, dass die Anzahl der in beiden Clustern vorhandenen Daten 2 beträgt. Dabei handelt es sich um die Daten d und e. Also ist $n_{22} = 2$. Diese Anzahlen lassen sich für alle möglichen Kombinationen von i und j bestimmen. Das führt zu Tabelle C.1 auf S. 576.

1 Vgl. Rand (1971), S. 847.
2 Vgl. Rand (1971), S. 847.

Tabelle C.1: Übersicht über die n_{ij}-Werte für zwei Clusterings Y und Y'

Y_j'	Y_i		$\lvert Y_j'\rvert$
	Y_1	Y_2	
Y_1'	2	0	2
Y_2'	1	2	3
Y_3'	0	1	1
$\lvert Y_i\rvert$	3	3	6

In der am weitesten rechts stehenden Spalte befinden sich dabei die Summen der Anzahlen der jeweiligen Zeile, d.h., in Cluster Y_3' ist insgesamt ein Element vorhanden. Auch die letzte Zeile der Tabelle beinhaltet Summen - die der darüberstehenden Spalten. D.h., sowohl in Cluster Y_1 als auch in Cluster Y_2 sind drei Elemente vorhanden. Ganz rechts unten befindet sich die Gesamtanzahl der Daten: sechs Elemente. Die n_{ij}-Werte können aus den restlichen Tabellenzellen für die jeweiligen Clusterkombinationen abgelesen werden.

Zur Berechnung des Rand Index werden die Werte in die folgende Formel eingesetzt[1]:

$$\Gamma(Y,Y') = \frac{\left[\binom{n}{2} - \left[\frac{1}{2}\left\{\sum_{i=1}^{|Y|}(\sum_{j=1}^{|Y'|} n_{ij})^2 + \sum_{j=1}^{|Y'|}(\sum_{i=1}^{|Y|} n_{ij})^2\right\} - \sum_{i=1}^{|Y|}\sum_{j=1}^{|Y'|}(n_{ij})^2\right]\right]}{\binom{n}{2}} \tag{C.1}$$

Mit den Werten aus dem angegebenen Beispiel erhält man Folgendes:

$$\begin{aligned}
\Gamma(Y,Y') &= \frac{\left[\binom{6}{2} - \left[\frac{1}{2}\left\{\sum_{i=1}^{2}(\sum_{j=1}^{3} n_{ij})^2 + \sum_{j=1}^{3}(\sum_{i=1}^{2} n_{ij})^2\right\} - \sum_{i=1}^{2}\sum_{j=1}^{3}(n_{ij})^2\right]\right]}{\binom{6}{2}} \\
&= \frac{\left[\binom{6}{2} - \left[\frac{1}{2}\left\{(2+1+0)^2 + (0+2+1)^2 + (2+0)^2 + (1+2)^2 + (0+1)^2\right\} - \right.\right.}{\binom{6}{2}} \\
&\quad \underline{\left.\left.(2^2+1^2+0^2+0^2+2^2+1^2)\right]\right]} \\
&= \frac{\left[\binom{6}{2} - \left[\frac{1}{2}\left\{18+14\right\} - 10\right]\right]}{\binom{6}{2}}
\end{aligned}$$

1 Dabei wurde die Formel aus dem Originaldokument leicht angepasst.

$$
\begin{aligned}
&= \frac{[15 - [16 - 10]]}{15} \\
&= \frac{9}{15} \\
&= 0,6
\end{aligned}
$$

C.2 Herleitung der verwendeten Formel

Die in dieser Arbeit benutzte Formel verwendet die Konstrukte TP, FN, FP und TN. Die Berechnungsvorschrift für diese Anzahlen befinden sich im Kapitel 2.5.1 auf S. 74. Alle diese Werte beruhen auf $h(L_i, \omega_j)$. Dabei handelt es sich um die Anzahl der Dokumente in Cluster ω_j, die zur Klasse L_i gehören. Zur Bestimmung dieser Anzahlen wird eine Tabelle aufgestellt.
Auf das in Rand (1971) verwendete Beispiel bezogen, ergibt sich Folgendes:

- Es wird ein Clustering Y', hier Ω, mit einer vorher definierten Klassifikation mit der Menge der Klassen Y, hier L, verglichen.
- Es werden zwei Klassen L_1 und L_2 festgelegt. Die erste Klasse enthält drei Dokumente: a, b und c und die zweite ebenfalls drei: d, e und f.
- Beim Clustern werden drei Cluster ω_1, ω_2 und ω_3 erzeugt. Der erste Cluster enthält die Elemente a und b, der zweite c, d und e und der dritte f.

Bestimmt man die Anzahlen der Dokumente in Cluster ω_j mit $j \in \{1, 2, 3\}$, die zur Klasse L_i mit $i \in \{1, 2\}$ gehören, so bedeutet das bspw., dass sie für den Cluster ω_2 und die Klasse L_2 zwei beträgt. Die weiteren Anzahlen können der Tabelle C.2 entnommen werden.

Tabelle C.2: Übersicht über die $h(L_i, \omega_j)$-Werte für eine Klassifikation mit der Menge der Klassen L und ein Clustering Ω

ω_j	L_i		$\lvert\omega_j\rvert$
	L_1	L_2	
ω_1	2	0	2
ω_2	1	2	3
ω_3	0	1	1
$\lvert L_i\rvert$	3	3	6

Vergleicht man diese Tabelle mit derjenigen, die die n_{ij}-Werte enthält - siehe Tabelle C.1 auf S. 576 - so stellt man fest, dass sie identisch sind. Die n_{ij}-Werte sind also mit den $h(L_i, \omega_j)$-Werten gleichzusetzen.

Die Berechnung des Rand Index erfolgt in dieser Arbeit folgendermaßen:

$$RI = \frac{TP + TN}{TP + FP + FN + TN} \tag{C.2}$$

Zu zeigen ist:

$$RI = \Gamma(Y, Y') \tag{C.3}$$

Oder vielmehr:

$$\frac{TP + TN}{TP + FP + FN + TN} = \frac{\left[\binom{n}{2} - \left[\frac{1}{2}\left\{\sum_{i=1}^{|Y|}(\sum_{j=1}^{|Y'|} n_{ij})^2 + \sum_{j=1}^{|Y'|}(\sum_{i=1}^{|Y|} n_{ij})^2\right\} - \sum_{i=1}^{|Y|}\sum_{j=1}^{|Y'|}(n_{ij})^2\right]\right]}{\binom{n}{2}} \tag{C.4}$$

Bevor gezeigt werden kann, dass es sich um eine Gleichung handelt, müssen die Definitionen für TP, TN, FP und FN in Kapitel 2.5.1 auf S. 74 umgeformt werden. Zunächst erfolgt das für die TP.[1]

$$\begin{aligned} TP &= \sum_{i=1}^{|L|}\sum_{j=1}^{c}\binom{h(L_i, \omega_j)}{2} \\ &= \sum_{i=1}^{|L|}\sum_{j=1}^{c}\frac{h(L_i, \omega_j) * [h(L_i, \omega_j) - 1]}{2} \\ &= \sum_{i=1}^{|L|}\sum_{j=1}^{c}\frac{(h(L_i, \omega_j))^2 - h(L_i, \omega_j)}{2} \\ &= \sum_{i=1}^{|L|}\sum_{j=1}^{c}\frac{(h(L_i, \omega_j))^2}{2} - \frac{h(L_i, \omega_j)}{2} \end{aligned}$$

1 Die Umformung von der ersten Zeile zur zweiten erfolgt mit Hilfe von $\binom{n}{2} = \frac{n*(n-1)}{2}$, da $\binom{n}{2} = \frac{n!}{2!*(n-2)!} = \frac{n*(n-1)*(n-2)!}{2!*(n-2)!} = \frac{n*(n-1)}{2*1} = \frac{n*(n-1)}{2}$.

$$= \sum_{i=1}^{|L|} \sum_{j=1}^{c} \frac{(h(L_i, \omega_j))^2}{2} - \sum_{i=1}^{|L|} \sum_{j=1}^{c} \frac{h(L_i, \omega_j)}{2}$$
$$= \frac{1}{2} \sum_{i=1}^{|L|} \sum_{j=1}^{c} (h(L_i, \omega_j))^2 - \frac{1}{2} \sum_{i=1}^{|L|} \sum_{j=1}^{c} h(L_i, \omega_j)$$

Das Ergebnis dieser Umformung wird direkt für FP verwendet.

$$\begin{aligned}
FP &= \sum_{j=1}^{c} \binom{|\omega_j|}{2} - TP \\
&= \sum_{j=1}^{c} \left\{ \frac{|\omega_j| * [|\omega_j| - 1]}{2} \right\} - TP \\
&= \frac{1}{2} \sum_{j=1}^{c} \left\{ (|\omega_j|)^2 - |\omega_j| \right\} - TP \\
&= \frac{1}{2} \sum_{j=1}^{c} \left\{ \left(\underbrace{\sum_{i=1}^{|L|} h(L_i, \omega_j)}_{=|\omega_j|} \right)^2 - \underbrace{\sum_{i=1}^{|L|} h(L_i, \omega_j)}_{=|\omega_j|} \right\} - TP \\
&= \frac{1}{2} \sum_{j=1}^{c} \left\{ \left(\sum_{i=1}^{|L|} h(L_i, \omega_j) \right)^2 - \sum_{i=1}^{|L|} h(L_i, \omega_j) \right\} - \left[\frac{1}{2} \sum_{i=1}^{|L|} \sum_{j=1}^{c} (h(L_i, \omega_j))^2 - \right. \\
&\quad \left. \frac{1}{2} \sum_{i=1}^{|L|} \sum_{j=1}^{c} h(L_i, \omega_j) \right] \\
&= \frac{1}{2} \sum_{j=1}^{c} \left(\sum_{i=1}^{|L|} h(L_i, \omega_j) \right)^2 - \frac{1}{2} \sum_{j=1}^{c} \sum_{i=1}^{|L|} h(L_i, \omega_j) - \frac{1}{2} \sum_{i=1}^{|L|} \sum_{j=1}^{c} (h(L_i, \omega_j))^2 + \\
&\quad \frac{1}{2} \sum_{i=1}^{|L|} \sum_{j=1}^{c} h(L_i, \omega_j) \\
&= \frac{1}{2} \sum_{j=1}^{c} \left(\sum_{i=1}^{|L|} h(L_i, \omega_j) \right)^2 - \frac{1}{2} \sum_{i=1}^{|L|} \sum_{j=1}^{c} (h(L_i, \omega_j))^2
\end{aligned}$$

Auch FN muss umgeformt werden:

$$
\begin{aligned}
FN &= \sum_{i=1}^{|L|} \binom{|L_i|}{2} - TP \\
&= \sum_{i=1}^{|L|} \left\{ \frac{|L_i| * [|L_i| - 1]}{2} \right\} - TP \\
&= \frac{1}{2} \sum_{i=1}^{|L|} \left\{ (|L_i|)^2 - |L_i| \right\} - TP \\
&= \frac{1}{2} \sum_{i=1}^{|L|} \left\{ \left(\underbrace{\sum_{j=1}^{c} h(L_i, \omega_j)}_{=|L_i|} \right)^2 - \underbrace{\sum_{j=1}^{c} h(L_i, \omega_j)}_{=|L_i|} \right\} - TP \\
&= \frac{1}{2} \sum_{i=1}^{|L|} \left\{ \left(\sum_{j=1}^{c} h(L_i, \omega_j) \right)^2 - \sum_{j=1}^{c} h(L_i, \omega_j) \right\} - \left[\frac{1}{2} \sum_{i=1}^{|L|} \sum_{j=1}^{c} \left(h(L_i, \omega_j) \right)^2 - \right. \\
&\qquad \left. \frac{1}{2} \sum_{i=1}^{|L|} \sum_{j=1}^{c} h(L_i, \omega_j) \right] \\
&= \frac{1}{2} \sum_{i=1}^{|L|} \left(\sum_{j=1}^{c} h(L_i, \omega_j) \right)^2 - \frac{1}{2} \sum_{i=1}^{|L|} \sum_{j=1}^{c} h(L_i, \omega_j) - \frac{1}{2} \sum_{i=1}^{|L|} \sum_{j=1}^{c} \left(h(L_i, \omega_j) \right)^2 + \\
&\qquad \frac{1}{2} \sum_{i=1}^{|L|} \sum_{j=1}^{c} h(L_i, \omega_j) \\
&= \frac{1}{2} \sum_{i=1}^{|L|} \left(\sum_{j=1}^{c} h(L_i, \omega_j) \right)^2 - \frac{1}{2} \sum_{i=1}^{|L|} \sum_{j=1}^{c} \left(h(L_i, \omega_j) \right)^2
\end{aligned}
$$

Abschließend wird noch TN umgeformt.

$$
\begin{aligned}
TN &= \binom{|D|}{2} - (TP + FN + FP) \\
&= \binom{|D|}{2} - TP - FN - FP
\end{aligned}
$$

$$
\begin{aligned}
&= \binom{|D|}{2} - \left[\frac{1}{2}\sum_{i=1}^{|L|}\sum_{j=1}^{c}\left(h(L_i,\omega_j)\right)^2 - \frac{1}{2}\sum_{i=1}^{|L|}\sum_{j=1}^{c}h(L_i,\omega_j)\right] - \\
&\quad \left[\frac{1}{2}\sum_{i=1}^{|L|}\left(\sum_{j=1}^{c}h(L_i,\omega_j)\right)^2 - \frac{1}{2}\sum_{i=1}^{|L|}\sum_{j=1}^{c}\left(h(L_i,\omega_j)\right)^2\right] - \\
&\quad \left[\frac{1}{2}\sum_{j=1}^{c}\left(\sum_{i=1}^{|L|}h(L_i,\omega_j)\right)^2 - \frac{1}{2}\sum_{i=1}^{|L|}\sum_{j=1}^{c}\left(h(L_i,\omega_j)\right)^2\right] \\
&= \binom{|D|}{2} - \frac{1}{2}\sum_{i=1}^{|L|}\sum_{j=1}^{c}\left(h(L_i,\omega_j)\right)^2 + \frac{1}{2}\sum_{i=1}^{|L|}\sum_{j=1}^{c}h(L_i,\omega_j) - \\
&\quad \frac{1}{2}\sum_{i=1}^{|L|}\left(\sum_{j=1}^{c}h(L_i,\omega_j)\right)^2 + \frac{1}{2}\sum_{i=1}^{|L|}\sum_{j=1}^{c}\left(h(L_i,\omega_j)\right)^2 - \\
&\quad \frac{1}{2}\sum_{j=1}^{c}\left(\sum_{i=1}^{|L|}h(L_i,\omega_j)\right)^2 + \frac{1}{2}\sum_{i=1}^{|L|}\sum_{j=1}^{c}\left(h(L_i,\omega_j)\right)^2 \\
&= \binom{|D|}{2} + \frac{1}{2}\sum_{i=1}^{|L|}\sum_{j=1}^{c}\left(h(L_i,\omega_j)\right)^2 + \frac{1}{2}\sum_{i=1}^{|L|}\sum_{j=1}^{c}h(L_i,\omega_j) - \\
&\quad \frac{1}{2}\sum_{i=1}^{|L|}\left(\sum_{j=1}^{c}h(L_i,\omega_j)\right)^2 - \frac{1}{2}\sum_{j=1}^{c}\left(\sum_{i=1}^{|L|}h(L_i,\omega_j)\right)^2
\end{aligned}
$$

Um zu zeigen, dass die Gleichung C.4 gilt, wird sie zunächst in zwei Teile - den Zähler und den Nenner - zerlegt.

$$
\frac{TP+TN}{TP+FP+FN+TN} = \frac{\left[\binom{n}{2} - \left[\frac{1}{2}\left\{\sum_{i=1}^{|Y|}(\sum_{j=1}^{|Y'|}n_{ij})^2 + \sum_{j=1}^{|Y'|}(\sum_{i=1}^{|Y|}n_{ij})^2\right\} - \sum_{i=1}^{|Y|}\sum_{j=1}^{|Y'|}(n_{ij})^2\right]\right]}{\binom{n}{2}}
$$

Zunächst wird bewiesen, dass die beiden Nenner gleich sind.[1]

$$
\begin{aligned}
TP+FP+FN+TN &= TP+FP+FN+\left(\binom{|D|}{2} - [TP+FP+FN]\right) \\
&= TP+FP+FN+\left(\binom{|D|}{2} - TP-FP-FN\right) \\
&= TP+FP+FN+\binom{|D|}{2} - TP-FP-FN
\end{aligned}
$$

1 Es gilt: $n == |D|$, siehe S. 577.

$$= \binom{|D|}{2} == \binom{n}{2}$$

Es folgt der Beweis für die Gleichheit der Zähler[1]:

$$\begin{aligned}
TP + TN = {} & \frac{1}{2}\sum_{i=1}^{|L|}\sum_{j=1}^{c}\left(h(L_i,\omega_j)\right)^2 - \frac{1}{2}\sum_{i=1}^{|L|}\sum_{j=1}^{c}h(L_i,\omega_j) + \binom{n}{2} + \\
& \frac{1}{2}\sum_{i=1}^{|L|}\sum_{j=1}^{c}\left(h(L_i,\omega_j)\right)^2 + \frac{1}{2}\sum_{i=1}^{|L|}\sum_{j=1}^{c}h(L_i,\omega_j) - \\
& \frac{1}{2}\sum_{i=1}^{|L|}\left(\sum_{j=1}^{c}h(L_i,\omega_j)\right)^2 - \frac{1}{2}\sum_{j=1}^{c}\left(\sum_{i=1}^{|L|}h(L_i,\omega_j)\right)^2 \\
= {} & \sum_{i=1}^{|L|}\sum_{j=1}^{c}\left(h(L_i,\omega_j)\right)^2 + \binom{n}{2} - \frac{1}{2}\sum_{i=1}^{|L|}\left(\sum_{j=1}^{c}h(L_i,\omega_j)\right)^2 - \\
& \frac{1}{2}\sum_{j=1}^{c}\left(\sum_{i=1}^{|L|}h(L_i,\omega_j)\right)^2 \\
= {} & \binom{n}{2} - \left[\frac{1}{2}\sum_{i=1}^{|L|}\left(\sum_{j=1}^{c}h(L_i,\omega_j)\right)^2 + \right. \\
& \left. \frac{1}{2}\sum_{j=1}^{c}\left(\sum_{i=1}^{|L|}h(L_i,\omega_j)\right)^2 - \sum_{i=1}^{|L|}\sum_{j=1}^{c}\left(h(L_i,\omega_j)\right)^2\right] \\
= {} & \binom{n}{2} - \left[\frac{1}{2}\left\{\sum_{i=1}^{|L|}\left(\sum_{j=1}^{c}h(L_i,\omega_j)\right)^2 + \sum_{j=1}^{c}\left(\sum_{i=1}^{|L|}h(L_i,\omega_j)\right)^2\right\} - \right. \\
& \left. \sum_{i=1}^{|L|}\sum_{j=1}^{c}\left(h(L_i,\omega_j)\right)^2\right] \\
= {} & \binom{n}{2} - \left[\frac{1}{2}\left\{\sum_{i=1}^{|L|}\left(\sum_{j=1}^{c}n_{ij}\right)^2 + \sum_{j=1}^{c}\left(\sum_{i=1}^{|L|}n_{ij}\right)^2\right\} - \sum_{i=1}^{|L|}\sum_{j=1}^{c}\left(n_{ij}\right)^2\right]
\end{aligned}$$

Damit ist bewiesen, dass Gleichung C.4 gilt.

1 Es gilt: $|Y| == |L|$ und $|Y'| == c$, siehe S. 577.

Anhang D

Beispieldurchlauf von Liste 1 für die Typ-L-Suffixe

Das Typ-L-Beispiel besteht aus dem Text T_2 =„stehplatz$“. Die Klassifizierung in Typ-S- und Typ-L-Suffixe sieht wie folgt aus:

T_2	s	t	e	h	p	l	a	t	z	$
Typ	S	L	S	S	L	L	S	S	L	L/S

Das Array A mit den eingezeichneten Buckets aufgrund der ersten Zeichen der Suffixe sieht wie folgt aus:

	1	2	3	4	5	6	7	8	9	10
A_{T_2}	10	7	3	4	6	5	1	2	8	9
	$	a	e	h	l	p	s	t	t	z

Es ergeben sich folgende L-Distanzen für die Suffixe:

	1	2	3	4	5	6	7	8	9	10
A_{T_2}	10	7	3	4	6	5	1	2	8	9
$Dist_{T_2}$	0	0	1	2	3	1	1	2	3	1

Aufgrund dieser Distanzen ergeben sich die folgenden Listen:

	0	1	2	3
1:	10	7	3	6
2:	4	8		
3:	5	9		

Das Array C mit eingezeichneten Buckets aufgrund des ersten Zeichens der Suffixe sieht wie folgt aus:

	1	2	3	4	5
C_{T_2}	10	6	5	2	9

Versieht man die m Listen von T_2 mit Buckets, ergibt sich folgendes Bild:

1:	10	7	3	6
2:	4	8		
3:	5	9		

Die beiden Hilfsarrays R und $lptr$ sehen wie folgt aus:

	1	2	3	4	5	6	7	8	9	10
R_{T_2}	-1	4	-1	-1	3	2	-1	-1	5	1

	1	2	3	4	5
$lptr_{T_2}$	1	2	3	4	5

Für den Text T_2 sieht der Sortierungsablauf nach der ersten Liste wie folgt aus[1]:

j = 1; **k = 4**; **merker = 4**
Liste j noch nicht vollständig durchlaufen

Bucketgrenze noch nicht erreicht
$i = j[k] = j[4] = 6$
$t = i - j = 6 - 1 = 5$
Bucket auslesen: $x = R_{T_2}[t] = R_{T_2}[5] = 3$
aktuelles Bucketende auslesen: $y = lptr_{T_2}[x] = lptr_{T_2}[3] = 3$
$\Rightarrow$ aktuelles Ende um Eins verringern: $lptr_{T_2}[3] = y - 1 = 2$
$k = k - 1 = 3$

Bucketgrenze erreicht
k = merker = 4

Bucketgrenze noch nicht erreicht
$i = j[k] = j[4] = 6$
$t = i - j = 6 - 1 = 5$
Bucket auslesen: $x = R_{T_2}[t] = R_{T_2}[5] = 3$
aktuelles Bucketende auslesen: $y = lptr_{T_2}[x] = lptr_{T_2}[3] = 2$
$z = y + 1 = 2 + 1 = 3$
$\Rightarrow$ neues Bucket zuweisen: $R_{T_2}[5] = z = 3$
$q = lptr_{T_2}[z] = lptr_{T_2}[3] = 2$

1 Wie bereits auf S. 129 der vorliegenden Arbeit erläutert, bräuchte die Sortierung hier nicht durchgeführt zu werden, da die Typ-L-Suffixe bereits sortiert vorliegen. Der Ablauf wird lediglich aus Gründen der Vollständigkeit und Anschaulichkeit gezeigt.

$\Rightarrow$ Bucketende vorhanden,
deshalb Bucketende um Eins erhöhen:
$lptr_{T_2}[3] = q + 1 = 3$
$k = k - 1 = 3$

Bucketgrenze erreicht
merker = k = 3

Liste j noch nicht vollständig durchlaufen

Bucketgrenze noch nicht erreicht
$i = j[k] = j[3] = 3$
$t = i - j = 3 - 1 = 2$
Bucket auslesen: $x = R_{T_2}[t] = R_{T_2}[2] = 4$
aktuelles Bucketende auslesen: $y = lptr_{T_2}[x] = lptr_{T_2}[4] = 4$
$\Rightarrow$ aktuelles Ende um Eins verringern: $lptr_{T_2}[4] = y - 1 = 3$
$k = k - 1 = 2$

Bucketgrenze erreicht
k = merker = 3

Bucketgrenze noch nicht erreicht
$i = j[k] = j[3] = 3$
$t = i - j = 3 - 1 = 2$
Bucket auslesen: $x = R_{T_2}[t] = R_{T_2}[2] = 4$
aktuelles Bucketende auslesen: $y = lptr_{T_2}[x] = lptr_{T_2}[4] = 3$
$z = y + 1 = 3 + 1 = 4$
$\Rightarrow$ neues Bucket zuweisen: $R_{T_2}[2] = z = 4$
$q = lptr_{T_2}[z] = lptr_{T_2}[4] = 3$
$\Rightarrow$ Bucketende vorhanden,
deshalb Bucketende um Eins erhöhen:
$lptr_{T_2}[4] = q + 1 = 4$
$k = k - 1 = 2$

Bucketgrenze erreicht
merker = k = 2

Liste j noch nicht vollständig durchlaufen

Bucketgrenze noch nicht erreicht
$i = j[k] = j[2] = 7$
$t = i - j = 7 - 1 = 6$
Bucket auslesen: $x = R_{T_2}[t] = R_{T_2}[6] = 2$

aktuelles Bucketende auslesen: $y = lptr_{T_2}[x] = lptr_{T_2}[2] = 2$

$\Rightarrow$ aktuelles Ende um Eins verringern: $lptr_{T_2}[2] = y - 1 = 1$

$k = k - 1 = 1$

Bucketgrenze erreicht

k = merker = 2

Bucketgrenze noch nicht erreicht

$i = j[k] = j[2] = 7$

$t = i - j = 7 - 1 = 6$

Bucket auslesen: $x = R_{T_2}[t] = R_{T_2}[6] = 2$

aktuelles Bucketende auslesen: $y = lptr_{T_2}[x] = lptr_{T_2}[2] = 1$

$z = y + 1 = 1 + 1 = 2$

$\Rightarrow$ neues Bucket zuweisen: $R_{T_2}[6] = z = 2$

$q = lptr_{T_2}[z] = lptr_{T_2}[2] = 1$

$\Rightarrow$ Bucketende vorhanden,
deshalb Bucketende um Eins erhöhen:

$lptr_{T_2}[2] = q + 1 = 2$

$k = k - 1 = 1$

Bucketgrenze erreicht

merker = k = 1

Liste j noch nicht vollständig durchlaufen

Bucketgrenze noch nicht erreicht

$i = j[k] = j[1] = 10$

$t = i - j = 10 - 1 = 9$

Bucket auslesen: $x = R_{T_2}[t] = R_{T_2}[9] = 5$

aktuelles Bucketende auslesen: $y = lptr_{T_2}[x] = lptr_{T_2}[5] = 5$

$\Rightarrow$ aktuelles Ende um Eins verringern: $lptr_{T_2}[5] = y - 1 = 4$

$k = k - 1 = 0$

Bucketgrenze erreicht

k = merker = 1

Bucketgrenze noch nicht erreicht

$i = j[k] = j[1] = 10$

$t = i - j = 10 - 1 = 9$

Bucket auslesen: $x = R_{T_2}[t] = R_{T_2}[9] = 5$

aktuelles Bucketende auslesen: $y = lptr_{T_2}[x] = lptr_{T_2}[5] = 4$

$z = y + 1 = 4 + 1 = 5$

$\Rightarrow$ neues Bucket zuweisen: $R_{T_2}[9] = z = 5$

$q = lptr_{T_2}[z] = lptr_{T_2}[5] = 4$

$\Rightarrow$ Bucketende vorhanden, deshalb Bucketende um Eins erhöhen:

$lptr_{T_2}[5] = q + 1 = 5$

$k = k - 1 = 0$

Bucketgrenze erreicht

merker = k = 0

Liste j vollständig durchlaufen

Damit ist der beispielhafte Durchlauf durch die erste der m Listen der Typ-L-Suffixe des Beispiels beendet.

Anhang E

Verwendete Elemente der Unified Modeling Language und ihre Bedeutung

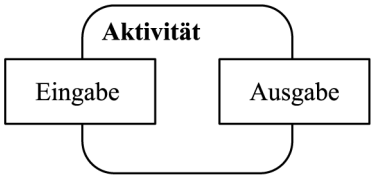
Im Folgenden werden die in der vorliegenden Arbeit verwendeten UML-Elemente grafisch dargestellt und erläutert. Dabei wird das vorgestellte UML-Element zunächst abgebildet und anschließend erläutert.

Eine Aktion ist in der Unified Modeling Language (UML) ein Element, um Verhalten darzustellen. Es handelt sich um die kleinste mögliche Einheit, um dieses zu tun.[1] Aktionen tauchen innerhalb von *Aktivitäten* - siehe weiter unten - auf und dienen der Darstellung einer ausführbaren Handlung innerhalb des größeren Gesamtkontextes einer Aktivität. In der vorliegenden Arbeit werden sie verwendet, um einzelne Schritte innerhalb einer Operation der Implementierung zu beschreiben und durch die Reihenfolge der Aktionen den Ablauf der Operation zu verdeutlichen. Die Beschreibung, was die Aktion durchführt, wird durch das Ersetzen des Textes „Aktion“ in der grafischen Darstellung ermöglicht.

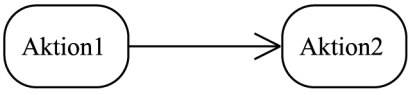
Die Aktivität beschreibt in der vorliegenden Arbeit eine Operation. Allgemein bildet sie den Kontext für Aktionen, verbindet also zusammengehörende Aktionen und bringt sie insbesondere in eine Abfolge.[2] Das unterstützen die weiter unten erläuterten *Kanten* und *Kontrollstrukturen*. Durch die Darstellung der implementierten

1 Vgl. Staud (2010), S. 121.
2 Vgl. Staud (2010), S. 129.

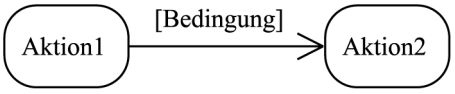
Operationen als Aktivitäten wird der Aufruf von weiteren Operationen durch Aktivitäten innerhalb von Aktivitäten dargestellt. Der Text „Aktivität“ in der grafischen Darstellung wird in der vorliegenden Arbeit durch den jeweiligen Operationsnamen ersetzt.



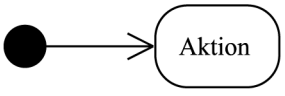
Jede Aktivität kann mit so genannten „Parameterknoten“[1] versehen werden. Die mit „Eingabe“ bezeichneten Knoten sind in der vorliegenden Arbeit Parameter, die der Operation übergeben werden und dementsprechend in ihr benutzt werden können. Die Parameter, die mit „Ausgabe“ bezeichnet sind, sind Parameter, die die Operation zurückgibt. Es können mehrere Eingabe- und Ausgabeparameter vorhanden sein, es muss jedoch keiner von beiden existieren.



Die gerichtete Kante zwischen zwei Aktionen zeigt den Kontrollfluss in der Aktivität an.[2] Dabei wird die Reihenfolge der Ausführung der Aktionen verdeutlicht. Die Aktion, an der die Kante beginnt, wird vor der Aktion, an der die Kante mit dem Pfeil endet, ausgeführt. In der vorliegenden Arbeit wird innerhalb der dargestellten Operation die Aktion1 vor der Aktion2 ausgeführt.

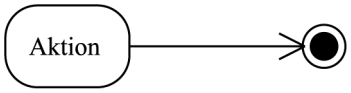


Diese Kantenart entspricht der zuvor genannten. Zusätzlich kann hier eine Bedingung angegeben werden, die erfüllt sein muss, damit diese Kante verfolgt und die nachfolgende Aktion ausgeführt wird.

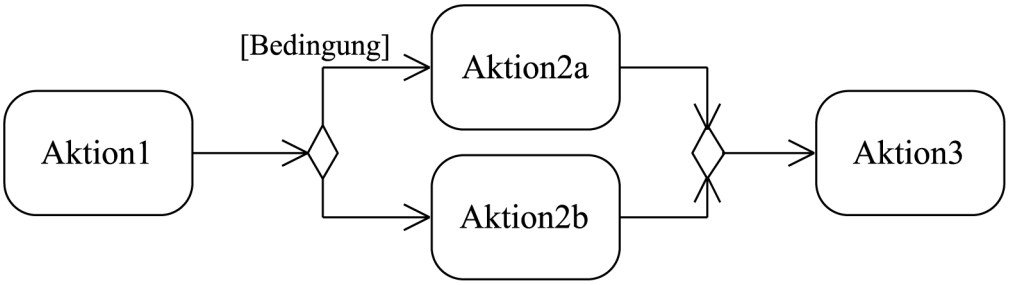


Der Startpunkt kennzeichnet den Anfang der Aktivität. Es kann einen oder auch mehrere geben.[3] Die darauf folgende Aktion, die durch eine gerichtete Kante mit dem Startpunkt verbunden ist, wird als erste Aktion innerhalb der Operation ausgeführt.

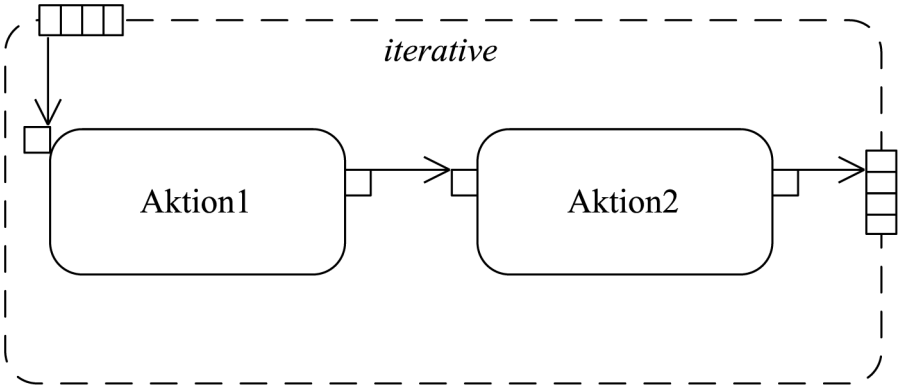
1 Staud (2010), S. 136 f.
2 Vgl. Staud (2010), S. 137.
3 Vgl. Staud (2010), S. 131.



Jede Aktivität und damit auch jede Operation endet mit einem oder mehreren Schlusspunkten.[1] Vom Schlusspunkt ausgehend existiert keine weitere Kante. In ihn können nur Kanten hineinlaufen.



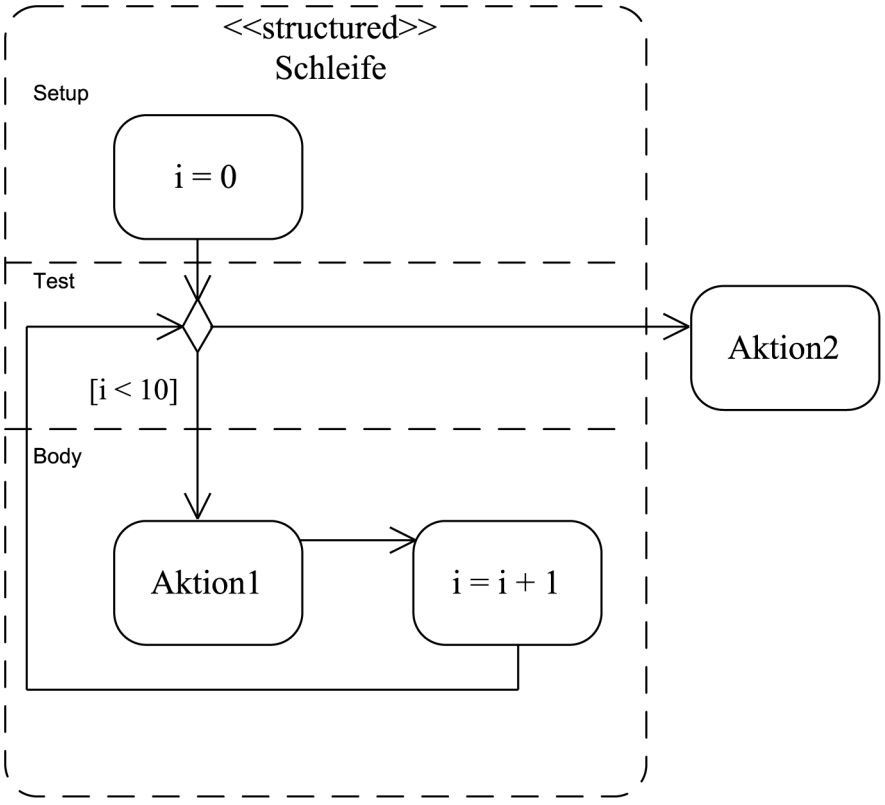
Wie bereits weiter oben erwähnt, existieren Kanten, die mit einer Bedingung versehen werden können. Die Kanten mit Bedingung[2] werden bei „Verzweigungen“[3] eingesetzt. Sie stellen den Punkt dar, an dem innerhalb der Operation entweder eine Aktion oder eine andere Aktion ausgeführt wird. Dabei muss die angegebene Bedingung erfüllt sein, um Aktion2a auszuführen, ansonsten wird Aktion2b ausgeführt. Nachdem die jeweilige Aktion - es können auch mehrere sein - ausgeführt ist, kann die Verarbeitung wieder „gradlinig“ weiterlaufen. Das wird durch eine „Zusammenführung“[4] verdeutlicht.



Sollen Aktionen mehrfach ausgeführt werden, so verwendet man Schleifen zur Darstellung. In der vorliegenden Arbeit werden zwei Typen verwendet: Zum einen eine

1 Vgl. Staud (2010), S. 131.
2 Steht in der vorliegenden Arbeit an der anderen Kante, die in der gleichen Verzweigung ihren Ausgangspunkt hat, keine Bedingung, so ist immer das Gegenteil der Bedingung, die angegeben ist, gemeint.
3 Staud (2010), S. 132.
4 Staud (2010), S. 132.

Schleife, die für jedes Element aus einer Menge von Elementen ausgeführt wird - das ist die hier beschriebene - und zum anderen eine Schleife, die ausgeführt wird, solange eine bestimmte Bedingung erfüllt ist - diese wird weiter unten erläutert.
Die Darstellung der Schleife erfolgt mit Hilfe einer so genannten „expansion region“[1]. Die Menge der Elemente wird mit „expansion nodes“[2] dargestellt. In der vorliegenden Arbeit werden zusätzlich, um zu verdeutlichen, dass die Schleife für jedes Element der Menge ausgeführt wird, die Aktionen innerhalb der Schleife mit „Pins“[3] versehen. Diese verdeutlichen eine Eingabe eines Elements in eine Aktion und die Ausgabe des Elements aus der Aktion. Der Text „iterative“ dient ebenfalls zur Verdeutlichung der Ausführungsart.



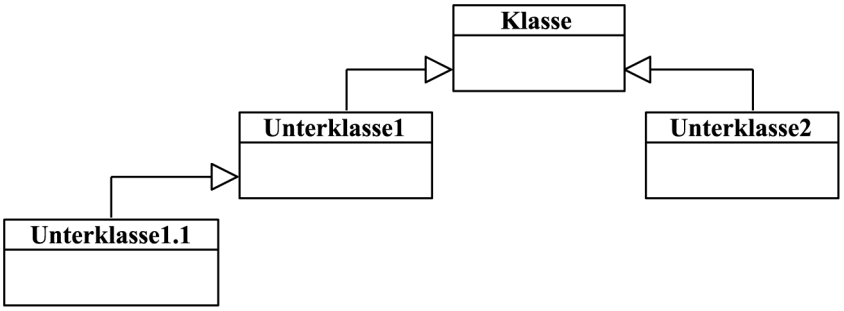
Die letzte in dieser Arbeit verwendete Kontrollstruktur ist die einer Schleife mit Bedingung. Dafür wird ein „loop node with regions“[4] verwendet. Die Angabe „«structured »“ bedeutet, dass es sich um eine strukturierte Schleife handelt, die aus drei Teilen besteht:

- Setup
  In dieser „Region“ des Knotens erfolgt die Initalisierung der Schleife.

1 Visual Paradigm (2012), ohne Seite.
2 Visual Paradigm (2012), ohne Seite.
3 Staud (2010), S. 124.
4 Visual Paradigm (2012), ohne Seite.

- Test
  Beim Test wird überprüft, ob eine Bedingung eingehalten wird. Ist das der Fall, so werden die Aktionen im Body ausgeführt, ansonsten wird die Schleife abgebrochen und mit der nächsten Aktion fortgefahren. Der Test wird jedes Mal durchgeführt, bevor die Aktionen innerhalb des Bodies erneut ausgeführt werden.

- Body
  Innerhalb des Bodies werden die Aktionen dargestellt, die innerhalb der Schleife durchgeführt werden sollen. Innerhalb des Bodies kann wiederum eine Schleife enthalten sein, d.h., es ist möglich, die Schleifen zu schachteln.



In der vorliegenden Arbeit wird der in den Experimenten verwendete Datensatz grafisch dargestellt. Das erfolgt mit Hilfe der UML-Elemente „Klasse“ und „Generalisierung“. In der Abbildung umfasst die ganz oben stehende und mit „Klasse“ bezeichnete Klasse die gesamte Klassifikation. Diese Klasse bildet also die Menge aller darin enthaltenen Klassen. Diese dort enthaltenen Klassen sind in der Abbildung mit der Bezeichnung „Unterklasse“ und einer Nummer versehen. Die Kanten mit dem Pfeil auf die jeweilige Oberklasse bedeuten, dass die Klassen in Pfeilrichtung generalisiert werden. Die Klassen an dem Ende der Kante ohne Pfeil sind also Spezialisierungen der Oberklassen.

Anhang F

Software-Prototyp

Neben der Beschreibung der Implementierung des Software-Prototyps, die in den Kapiteln 3.1.4.4 ab S. 148, 3.1.5.3 ab S. 198, 3.2.3 ab S. 222, 4 ab S. 229 und 5 ab S. 458 erfolgt ist, befindet sich der Quellcode der Implementierung auf der beigefügten CD.
Einen Überblick über die auf der CD enthaltenen Dateien befindet sich in der Datei „ReadMe.txt“ auf der beigefügten CD.